ELSEVIER

Theoretical Computer Science 240 (2000) 215–254

Theoretical Computer Science

www.elsevier.com/locate/tcs

Rocco De Nicola[a,*], GianLuigi Ferrari[b], Rosario Pugliese[a], Betti Venneri[a]

[a] *Dipartimento di Sistemi e Informatica, Università di Firenze, Via Lombruso 6/17, I-50134 Firenze, Italy*

[b] *Dipartimento di Informatica, Università di Pisa, Corso Italia 40, I-56100, Pisa, Italy*

Abstract

KLAIM is an experimental programming language that supports a programming paradigm where both processes and data can be moved across different computing environments. This paper presents the mathematical foundations of the KLAIM type system; this system permits checking access rights violations of mobile agents. Types are used to describe the intentions (read, write, execute, ...) of processes relative to the different localities with which they are willing to interact, or to which they want to migrate. Type checking then determines whether processes comply with the declared intentions, and whether they have been assigned the necessary rights to perform the intended operations at the specified localities. The KLAIM type system encompasses both subtyping and recursively defined types. The former occurs naturally when considering hierarchies of access rights, while the latter is needed to model migration of recursive processes. © 2000 Elsevier Science B.V. All rights reserved.

Keywords: Network aware programming; Mobile agents; Coordination; Type systems; Access control

1. Introduction

Network computing is calling for new programming paradigms and for new programming languages that model interactions among clients and servers by means of *mobile agents*; these are programs that are transported and executed on different hosts. Security, i.e. *privacy* and *integrity* of data, is a key issue in the development of mobile applications. One can easily imagine malicious mobile agents attempting to access private information, or modifying private data. Hence, a server receiving a mobile

☆ This work has been partially supported by Esprit Working Groups *CONFER2* and *COORDINA*, and by CNR Projects Modelli e Metodi per la Matematica e l'Ingegneria and Metodologie e Strumenti di Analisi, Verifica e Validazione di Sistemi Software Affidabili.

* Corresponding author.

E-mail addresses: denicola@dsi.unifi.it (R. De Nicola), giangi@di.unipi.it (G.-L. Ferrari), pugliese@dsi.unifi.it (R. Pugliese), venneri@dsi.unifi.it (B. Venneri)

PII: S0304-3975(99)00232-7

agent for execution needs to impose strong requirements to ensure that the incoming agent does not violate privacy and jeopardize the integrity of the information. Similarly, mobile agents need tools to ensure that their execution at the server site does not compromise their integrity or security.

Programming languages for mobile agents are based on policies (both at compilation and run time) that restrict privileges and capabilities more than needed. This may unnecessarily reduce the expressive power (and the capabilities) of mobile agents. Moreover, it might not be obvious how to guarantee that certain desired security properties are enforced by the language implementation. More generally, there is a lack of formal foundations to express and prove desired security properties of programs.

Recently, several researchers have explored the possibility of considering security issues at the level of language design aiming at embedding protection mechanisms in the languages. For instance, the language Java [7] exploits type information as a foundation of its security: well-typed Java programs (and the corresponding verified bytecode) will never compromise the integrity of certain data. In the area of functional programming, *type systems* are successfully used to avoid programming errors (type safety) by means of checks at compile time. Types are used to define the notion of *well-behaved* programs, and only programs that comply with the requirement of the type system are executed.

In this paper we discuss the design of the security type system for KLAIM (*a Kernel Language for Agents Interaction and Mobility*) [18], an experimental programming language specifically designed for programming mobile agents. KLAIM uses types to protect resources and data and to express and enforce policies for access control. The type system is used to guarantee that the operations that processes intend to perform at various network sites comply with the processes' access rights.

KLAIM supports a programming paradigm where programs can migrate from one computing environment to another. The language consists of core Linda [22, 15] with multiple located tuple spaces and of a set of process operators, borrowed from Milner's CCS [28]. KLAIM tuple spaces and processes are distributed over different localities, which are considered as first-class data. Linda operations are indexed with the locations of the tuple space they operate on. This allows programmers to distribute/retrieve data and processes over/from different nodes directly. Programmers share their control with what we call the *net coordinators*. Net coordinators describe the distributed infrastructure necessary for managing physical distribution of processes, allocation policies, and agents mobility.

Let us now see how a system composed of a process *Server* and two identical processes *Client* can be programmed in KLAIM . Afterwards, we shall see how types can be used to specify and enforce access control polices. The server process is programmed in KLAIM as follows:

$$Server \stackrel{\text{def}}{=} \mathbf{out}(l)@\mathbf{self}.\mathbf{nil}.$$

Server first adds a tuple that contains the locality l (**out**(l)) to its local tuple space (@**self**), then evolves to the terminated process **nil**. The client process is programmed

as follows:

$$Client \stackrel{\text{def}}{=} \mathbf{read}(!\,u)@l_S.\mathbf{eval}(P)@u.\mathbf{nil}.$$

Client first accesses the tuple space located at l_S to read an address u ($\mathbf{read}(!\,u)@l_S$, where $!\,u$ is a formal variable), then sends process P for execution at u ($\mathbf{eval}(P)@u$), finally evolves to **nil**.

A net coordinator might allocate *Server* on site s (this implies that `self` is bound to s) and the two processes *Client* on sites s_1 and s_2. At both these sites, care is taken that l_S is bound to s; this allows clients to interact with the server.

KLAIM types provide information about the intentions of processes: downloading/consuming a tuple, producing a tuple, activating a process, and creating a new tuple space. We will use $\{r,i,o,e,n\}$ to indicate the set of capabilities, where each symbol stands for the operation whose name begins with it; r denotes the capability of executing a **read** operation, i stands for **in**, o for **out**, e for **eval**, and n for **newloc**. Semantically, types are functions mapping localities (and locality variables) into functions from sets of capabilities to types. The KLAIM type system encompasses both subtyping and recursively defined types. Subtyping naturally emerges when considering *hierarchies* of access rights. Recursive types are used for dealing with mobile recursive agents.

The typing analysis of KLAIM programs is structured in two phases reflecting the two-level syntax of KLAIM . The first phase deduces the intentions of processes relatively to the different localities they are willing to interact with or they want to migrate to. This is obtained by means of an inference system that statically assigns types to processes, and checks whether processes behave in accordance with their declared intentions. All rules of the system are syntax driven and such that for any process there exists a minimal type smaller than all the types deducible for it.

The second phase statically checks whether each process has the necessary rights to perform the intended operations, i.e. whether the process violates the access rights as granted by the net coordinator. Capabilities are used differently by processes and net coordinators. The capabilities associated to a locality (or to a locality variable) ℓ, within a process type carry information about the operations the process intends to perform at ℓ. Net coordinators use capabilities to specify the access policy of each site of a net in terms of access rights and execution privileges. In practice, net coordinators statically decorate each node of a net by a unique type that codifies the policy for controlling access from that node to the other nodes of the net. A net is called well-typed whenever, for any site, the types of the processes allocated on that site are not greater than the type of the site.

The role of the two phases is illustrated by the client/server system described above. If process P has type δ_P, the outcome of the first stage of typing analysis of *Client* is the type

$$\delta_c = l_S \mapsto \{r\} \mapsto \bot, u \mapsto \{e\} \mapsto \delta_P.$$

It states that *Client* intends to perform a **read** operation at locality l_S and intends to send a process with type δ_P for execution at the locality denoted by u.

Let us assume that the net coordinator grants the following access rights to the sites s_1 and s_2 where *Client* is duplicated:

$$\delta_{s_1} = s \mapsto \{r\} \mapsto \bot, u \mapsto \{e\} \mapsto \delta_P, \quad \delta_{s_2} = s \mapsto \{r\} \mapsto \bot,$$

then only the process located at s_1 has the right of sending processes with type δ_P for execution at u. Indeed, the net

$$s ::^{\delta_s}_{[s/\mathbf{self}]} \mathit{Client} \parallel s_1 ::^{\delta_{s_1}}_{[s_1/\mathbf{self}, s/l_S]} \mathit{Server} \parallel s_2 ::^{\delta_{s_2}}_{[s_2/\mathbf{self}, s/l_S]} \mathit{Server}$$

where δ_s is the type of site s, is not well-typed.

By relying on static typing and dynamic type checking, we prove that well-typedness is an invariant of the operational semantics (*subject reduction*) and that well-typed nets are free from run time errors caused by misuse of access rights (*type safety*). Dynamic type checking is necessary because the creation of new sites modifies the types of the other sites of the net and is useful for controlling (typed) data exchange.

This paper provides the mathematical foundations of the KLAIM type system. Its main technical contribution is a subject reduction theorem and a type safety theorem. We also prove the decidability of the type system and provide an algorithm to compute types of processes.

The rest of the paper is organized as follows. Section 2 introduces the syntax and the informal semantics of KLAIM processes, types and nets. Section 3 defines type equality and the subtyping relation, while Section 4 defines the type inference system for deriving process types and introduces the notion of well-typed net. Section 5 defines the operational semantics of KLAIM processes and nets. Section 6 states and proves the subject reduction theorem and the type safety theorem for our type system. Section 7 contains a few programming examples. For the sake of readability, detailed proofs of syntactical properties of the type system are postponed to Section 8. The final section contains comparisons with related work and hints for further research.

2. Klaim and its informal semantics

KLAIM consists of a core Linda with multiple tuple spaces and of a set of operators, borrowed from Milner's CCS [28]. A distinguishing feature is that tuples and operations over them are located at specific sites of a net and types are used to control access rights of processes over these sites. We start this section by summarizing the main features of Linda (the interested reader is referred to, e.g., [23, 16, 15] for more details). Then, we present the syntax of KLAIM (processes, types and nets).

Most of the presentation of the untyped part of the language is borrowed from [18]. There, we also outline the main features of the KLAIM type system without providing the actual syntax for types, the explicit notion of run time error and the proofs of the subject reduction theorem and of the type safety theorem.

Linda is a coordination language that relies on an asynchronous and associative communication mechanism based on a shared global environment called tuple space (TS). A tuple space is a multiset of tuples, that are sequences of actual fields, i.e.

expressions or values, and formal fields, i.e. variables. *Pattern matching* is used to select tuples in a TS. Two tuples match if they have the same number of fields and corresponding fields have matching values or variables; variables match any value of the same type, and two values match only if they are identical. Linda provides just four main primitives for handling tuples: two (non-blocking) operations add tuples to a TS, two (possibly blocking) operations access tuples in the TS.

The Linda asynchronous communication model allows programmers to explicitly control process interactions via shared data and to use the same set of primitives both for data manipulation and for process synchronization. This has the advantage of rendering explicit all the interactions of a program with its environment. The original Linda primitives are, however, not completely adequate for distributed programming; data protection and security, which are key features of mobile applications, are problematic because the Linda communication model cannot guarantee data privacy. Distribution and access control are the main concerns of our contribution.

2.1. Klaim processes

Hereafter, we shall exploit the syntactic categories listed below; all of them are followed by the symbols we will use (sometimes with indices) to refer to their elements.

$\mathcal{S}(s)$ is a set of *sites* (or *physical localities*), $Loc(l)$ is a set of (*logical*) *localities*, $Vloc(u)$ is a set of *locality variables*, $Val(v)$ is a set of basic *values*, $Var(x)$ is a set of *value variables*, $Exp(e)$ is the category of *value expressions*, $\chi(X)$ is a set of *process variables* and $\Psi(A)$ is a set of parameterized *process identifiers*.

A site can be considered as the *address* (or the *name*) of a node where processes and tuple spaces might be located. Localities are the symbolic names of sites and programmers are not required to know the precise mapping of localities on sites. Localities allow programmers to structure programs over distributed environments while ignoring the precise allocations of processes and data. A distinguished locality `self` ($\in Loc$) is assumed that programs can use to refer to their own execution site. The set of locality variables, *Vloc*, is partitioned into two subsets, *NVloc* and *TVloc*. Variables in *NVloc* are used to create new sites (i.e. as arguments of **newloc**), and variables in *TVloc* are used to bind localities (i.e. as formals of tuples). Expressions are built up from values and value variables, by using a set of operators (not specified here). Parameters of process identifiers can be processes, localities and values, and they are provided in this order.

We sometimes use ℓ to denote both localities and locality variables. Moreover, we write $\tilde{\cdot}$ to denote a sequence of objects and $\{\tilde{\cdot}\}$ to denote the set of objects in $\tilde{\cdot}$.

KLAIM *terms* are obtained from the abstract syntax in Table 1. Process and locality variables are typed whenever they are bound; for the sake of simplicity, value variables are kept untyped. The precise syntax of types, that are ranged over by δ, will be introduced in the next section.

Fields, ranged over by f, can be actual fields (i.e. expressions, processes, localities or locality variables) and formal fields. To avoid confusing names with formal fields

Table 1
Process syntax

P	::=	**nil**	(null process)
	|	$a.P$	(action prefixing)
	|	$P_1 \mid P_2$	(parallel composition)
	|	X	(process variable)
	|	$A\langle \tilde{P}, \tilde{\ell}, \tilde{e} \rangle$	(process invocation)
a	::=	$\mathbf{out}(t)@\ell \mid \mathbf{in}(t)@\ell \mid \mathbf{read}(t)@\ell \mid \mathbf{eval}(P)@\ell \mid \mathbf{newloc}(u:\delta)$	
t	::=	$f \mid f,t$	
f	::=	$e \mid P \mid \ell \mid x \mid X:\delta \mid u:\delta$	

these are denoted by "! *var*", where *var* is a generic variable. Tuples, ranged over by t, are sequences of fields; hence, for a tuple t, $\{t\}$ will denote the set of fields of t.

KLAIM processes may access tuple spaces through explicit naming: operations are indexed with the locality of the tuple space. The (non-blocking) operation $\mathbf{out}(t)@\ell$ adds the tuple resulting from the evaluation of t to the TS located at ℓ. Two (possibly blocking) operations, $\mathbf{in}(t)@\ell$ and $\mathbf{read}(t)@\ell$, access tuples in the TS located at ℓ. The operation $\mathbf{in}(t)@\ell$ evaluates t and looks for a matching tuple t' in the TS; if t' is found, it is removed from the TS. The corresponding values of t' are then assigned to the variables of t and the operation terminates. If no matching tuple is found, the operation is suspended until one is available. The operation $\mathbf{read}(t)@\ell$ differs from $\mathbf{in}(t)@\ell$ only in that the tuple t' selected by pattern matching is not removed from the TS.

New threads of executions are dynamically activated through the operation $\mathbf{eval}(P)@\ell$ that spawns a process (whose code is given by P) at the node named ℓ.

Processes can create new sites through the prefix $\mathbf{newloc}(u:\delta)$, that is not indexed with a locality because it is always executed at the current site (`self`). This operation creates a "fresh" site that can be accessed via the locality variable u, the type δ specifies the access control policy of this site.

The operators for building processes are commonly used in Process Algebras and aim at modelling basic behaviours of concurrent systems. The expression **nil** stands for the process that cannot perform any action, $a.P$ stands for the process that first executes action a and then behaves like P, $P_1 \mid P_2$ stands for the parallel composition of P_1 and P_2, and $A\langle \tilde{P}, \tilde{\ell}, \tilde{e} \rangle$ stands for the invocation of the process identified by A with actual parameters $\tilde{P}$, $\tilde{\ell}$ and $\tilde{e}$.

Variables occurring in process terms can be bound by prefixes and process defining equations. More precisely, prefixes $\mathbf{in}(t)@\ell._-$ and $\mathbf{read}(t)@\ell._-$ act as binders for variables in the formal fields of t. Prefix $\mathbf{newloc}(u:\delta)._-$ binds the locality variable u. Definition $A(\tilde{X}:\tilde{\delta}, \tilde{u}:\tilde{\delta}, \tilde{x}) \stackrel{\text{def}}{=} P$ is considered as a binder for the variables $\{\tilde{X}\} \cup \{\tilde{u}\}$

$\cup\{\tilde{x}\}$. Hereafter, we assume that all bound names in processes are distinct and require that the arguments of **eval** operations do not contain free process variables.

We will use the standard notation $P[e/x]$ to indicate the substitution of the value expression e for the free occurrences of the variable x in P; $P[\tilde{e}/\tilde{x}]$ will denote the simultaneous substitution of any free occurrence of $x \in \{\tilde{x}\}$ with the corresponding $e \in \{\tilde{e}\}$ in P. In particular, when substitutions refer to locality variables, like $P[\ell/u]$, they have to be applied also to the type specifications therein. Notation $P[\tilde{P}/\tilde{X}, \tilde{\ell}/\tilde{u}, \tilde{e}/\tilde{x}]$ has the expected meaning.

Process identifiers are used in recursive process definitions. It is assumed that each process identifier A has a *single* defining equation $A(\tilde{X}:\tilde{\delta}, \tilde{u}:\tilde{\delta}, \tilde{x}) \stackrel{\text{def}}{=} P$, where process and locality parameters are explicitly typed. All free (value, process and locality) variables of P are contained in $\{\tilde{X}, \tilde{u}, \tilde{x}\}$ and, to guarantee uniqueness of solution of recursive process definitions, all its process variables and identifiers are *guarded*, i.e. each process variable/identifier occurs within the scope of a blocking **in**/**read** prefix.

A *process* is a term without free variables; localities occurring in processes are considered as constants. In Section 2.3, we will see that they are names whose meaning is defined (i.e. mapped onto sites) by coordinators. Both processes and localities are first-class data and can be manipulated and generated like any other data occurring in tuples. Processes have *higher-order* capabilities and can be exchanged when communicating.

Differently from previous presentations [17, 18], the operator for explicit nondeterministic composition of processes is not considered. This will considerably simplify the operational semantics while leaving expressivity unchanged. Indeed, this allows us to have a single-level operational semantics instead of the two-level semantics presented in [17, 18]. There, we first considered the evolution of single processes and then that of whole nets. The two levels enabled us to determine which operand of a nondeterministic composition had been involved in the reduction step of a net. The absence of explicit nondeterministic choice does not influence expressivity because nondeterminism is inherent in the definition of KLAIM operations. It arises when more **in**/**read** operations are suspended while waiting for a tuple or when an **in**/**read** operation has more than one matching tuple. In the former case, when a matching tuple becomes available, only one of the suspended operations is nondeterministically selected to proceed; in the latter case, one of the matching tuples is arbitrarily chosen. Hence, within the reduced language, nondeterminism can be modelled via the pattern matching mechanism; but it could also be retrieved like, e.g., in [31].

There are several process calculi using explicit localities. For instance, localities in Dπ [25] model distribution and mobility of π-calculus processes. In the distributed Join calculus [21], another variant of π-calculus, channels have a unique locality and agents may move from a locality to any other locality. In [4] it has been shown that a fragment of the asynchronous π-calculus with localities captures the main features of the distributed Join calculus. In [6] localities are used to model failures in fragments of π-calculus. Finally, to model distribution of computations and resources over sites of networks, the Ambient calculus [13] and the Seal calculus [37] rely on the notion of execution environment (ambient) rather than on that of locality.

Table 2
Type syntax

δ	$::=$	$\bot$	(empty type)
	$\mid$	$\top$	(universal type)
	$\mid$	$\ell \mapsto \pi \mapsto \delta$	(arrow type)
	$\mid$	δ_1, δ_2	(union type)
	$\mid$	v	(type variable)
	$\mid$	$\mu v.\delta$	(recursive type)

2.2. Klaim types

We will use $\{r, i, o, e, n_U \mid U \subseteq_f NVloc, U \neq \emptyset\}$ to indicate the set of process *capabilities*, where each symbol stands for the operation whose name begins with it; r denotes the capability of executing a **read** operation, i the capability of executing an **in** operation, and so on. Capability n, that corresponds to the **newloc** operations, is indexed by U, a finite, non-empty set of locality variables which will be used to record the set of (references to) sites dinamically created at a given locality. *Polarities* are non-empty subsets of $\{r, i, o, e, n\}$, where n is indexed by U. Union between polarities is standard set union but $\{n_U\} \cup \{n_{U'}\} = \{n_{U \cup U'}\}$. We use Π, ranged over by π, to denote the set of all polarities. We will write n in place of n_U whenever U can be safely ignored.

KLAIM types are defined by the abstract syntax in Table 2; there v ranges over type variables and μ denotes the recursive operator. Hereafter, the following notational convention will be used: "$\mapsto$" binds stronger than "μ", that binds stronger than ",". From a semantical point of view, a *type* is a finite map that assigns functions from polarities to types to both localities and locality variables.

The type $\bot$ denotes "void", i.e. no intention is declared by the process, and, semantically, corresponds to the smallest type. Conversely, the type $\top$ denotes the intention of performing any kind of operations and is the greatest type. A type of the form $\ell \mapsto \pi \mapsto \delta$ describes the intention of performing the actions corresponding to the polarity π at ℓ, moreover it imposes constraint δ on the processes that could possibly be executed at ℓ. Hence, the arrow type operator has the usual meaning of logical implication. The type δ_1, δ_2 is the union of types δ_1 and δ_2; semantically, it is their *least upper bound*. Recursive types are used for typing migrating recursive processes.

A type δ generated from the grammar in Table 2 is such that any recursive type $\mu v.\delta'$ occurring in δ does not contain v on the left of $\mapsto$. This is a simplification of the notion of *positive* type of [5]. The fact that we only deal with a restricted form of types will be essential in the definition of subtyping.

A consequence of our requirement that each process variable/identifier in process definitions be guarded (i.e. occurs within the scope of a blocking **in**/**read** prefix) is the fact that we only use recursive types $\mu v.\delta$ whose body δ can be written as $\delta_1, \ldots, \delta_n$ and at least one of the δ_i has the form $\ell \mapsto \pi \mapsto \delta'$ (for some ℓ, π and δ' such that

$\{r,i\} \cap \pi \neq \emptyset$). In the following, we will only consider types that satisfy the condition above and are such that in types of the form $\ell \mapsto \pi \mapsto \delta$ if $e \notin \pi$ then $\delta = \bot$; they will be called *legal* types. Note that the syntax in Table 2 is less restrictive; it also permits types of the form $\mu v.v$.

The following notion will be useful in later proofs.

Definition 2.1. The set $Sub(\delta)$ of *subterms* of a type δ is the set of types inductively defined as follows:

- $Sub(\bot) = \{\bot\}$;
- $Sub(\top) = \{\top\}$;
- $Sub(v) = \{v\}$;
- $Sub(\ell \mapsto \pi \mapsto \delta) = \{\ell \mapsto \pi \mapsto \delta\} \cup Sub(\delta)$;
- $Sub(\delta_1, \ldots, \delta_n) = \{\delta_1, \ldots, \delta_n\} \cup Sub(\delta_1) \cup \cdots \cup Sub(\delta_n)$;
- $Sub(\mu v.\delta) = \{\mu v.\delta\} \cup \{\delta'[\mu v.\delta/v] \mid \delta' \in Sub(\delta)\}$.

Finiteness of the sets of subterms for recursive types has been proven in [12]; it also holds for $Sub(\delta)$.

In the rest of this paper we will extensively use systems of (possibly mutually recursive) type equations for defining n-tuples of types. The solution of a set of n *type equations*

$$\{v_1 = \delta_1, \ldots, v_n = \delta_n\},$$

where $v_1, \ldots, v_n$ are all the free type variables occurring in $\delta_1, \ldots, \delta_n$, is the n-tuple of types $\delta_1^*, \ldots, \delta_n^*$ obtained by standard iterative techniques (see, e.g. [36]).

Before ending the section, we would like to remark that recursive processes do not necessarily have recursive types. Recursive types arise when typing recursive processes that can migrate. For instance, let us consider the Client/Server system presented in the Introduction. A different client process can be programmed as follows:

$$Client_1 \stackrel{\text{def}}{=} \mathbf{read}(!\,u)@l_S.\mathbf{eval}(P)@u.Client_1.$$

Suppose, for simplicity, that P has no recursion and does not call (even indirectly) process $Client_1$. The type inferred for $Client_1$ will be δ_c, the same type as the one derived for process *Client* in the introduction. Instead, consider process

$$Client_2 \stackrel{\text{def}}{=} \mathbf{read}(!\,u)@l_S.\mathbf{eval}(P \mid Client_2)@u.\mathbf{nil}$$

and again suppose that P has no recursion and does not call (even indirectly) process $Client_2$. The type of process $Client_2$ now will be the solution of the following recursive type equation in the type variable v:

$$v = l_S \mapsto \{r\} \mapsto \bot,\ u \mapsto \{e\} \mapsto (\delta_P, v).$$

Intuitively, the resulting type states that $Client_2$ intends to perform **read** operations at l_S and to migrate to (the locality denoted by) u together with a process with type δ_P.

2.3. Klaim nets

A *node* is a 4-tuple $(s, P_s, \delta_s, \rho_s)$ where s is a site, P_s is the process located at s, δ_s is the type of s specifying the access control policy of s, and ρ_s is the *allocation environment* of s, i.e. a (partial) function from localities to sites. We write $s ::_{\rho_s}^{\delta_s} P_s$ to denote the node $(s, P_s, \delta_s, \rho_s)$.

Hereafter, $\mathscr{E}$ denotes the set of environments, ϕ the empty environment, and $\{s/l\}$ the environment that maps the locality l on the site s. We use $\ell\{\rho\}$ to denote $\rho(\ell)$, if $\rho(\ell)$ is defined, and ℓ, otherwise; moreover, $\rho[s/\ell]$ denotes the environment ρ' such that $\rho'(\ell) = s$ and $\rho'(\ell') = \rho(\ell')$ for $\ell' \neq \ell$.

To specify the mutual access policies of a set of nodes, hence to consistently assign types to sites/nodes, we make use of a partial function Λ that, for each site s, describes the access rights of s on the other sites. Function Λ is used to derive a system of type equations whose solution will give the types of the nodes. Types of nodes have the same syntax as types of processes. However, strictly speaking, the formers cannot be generated by the grammar given in Table 2, since we required ℓ to stand for localities and locality variables. For types of nodes, we let ℓ range over sites.

Definition 2.2. A *KLAIM net* is a pair $N_S : \Lambda$ where N_S is a finite multiset of nodes over the set of sites S, and $\Lambda : S \rightharpoonup (S \rightharpoonup \Pi)$ is a partial function. Function Λ induces a system of n type equations in the variables $\{v_{s_i} \mid s_i \in S\}$ where, if $\{s_1^i, \ldots, s_k^i\} = dom(\Lambda(s_i))$, the equation of site s_i is

$$v_{s_i} = s_1^i \mapsto \Lambda(s_i)(s_1^i) \mapsto v_{s_1^i}, \ldots, s_k^i \mapsto \Lambda(s_i)(s_k^i) \mapsto v_{s_k^i},$$

and the following conditions hold:

1. for any node $s ::_{\rho_s}^{\delta_s} P$, $\rho_s(self) = s$, and δ_s is the component of the solution of the system of type equations induced by Λ corresponding to s;
2. for any pair of nodes $s ::_{\rho_s}^{\delta_s} P$ and $s' ::_{\rho_{s'}}^{\delta_{s'}} P'$, $s = s'$ implies $\rho_s = \rho_{s'}$ and $\delta_s = \delta_{s'}$.

Condition 1. expresses that the local allocation environment of a node maps localities on the sites of the net; in particular, `self` is mapped on the name (site) of the node. Moreover, the type of the node is determined by the function Λ associated to the net. The idea underlying condition 2. is that the nodes of a net over the same site do have the same allocation environment and type as well.

In the following, we will write N instead of $N : \Lambda$ whenever unambiguous. Given a net N, we assume existence of a function st that returns all sites of N and write N_S for a net N such that $st(N) = S$. Nets will also be written according to the syntax in Table 3. This notation will be used in Section 5 in giving the operational semantics of nets.

3. Type equality and subtyping

This section introduces the equality relation on types and the subtyping relation.

Table 3
Net syntax

N	$::=$	$s ::^{\delta}_{\rho} P$	(node)
	$\mid$	$N_1 \parallel N_2$	(net composition)

Table 4
Type equalities

Equalities on union types

(1) $\delta_1, \delta_2 = \delta_2, \delta_1$

(2) $(\delta_1, \delta_2), \delta_3 = \delta_1, (\delta_2, \delta_3)$

(3) $\delta, \delta = \delta$

(4) $\bot, \delta = \delta$

(5) $\top, \delta = \top$

(6) $\ell \mapsto \pi_1 \mapsto \delta_1, \ell \mapsto \pi_2 \mapsto \delta_2 = \ell \mapsto \pi_1 \cup \pi_2 \mapsto (\delta_1, \delta_2)$

Equalities on recursive types

(7) $\mu v_1.\delta_1, \mu v_2.\delta_2 = \mu v.(\delta_1[v/v_1], \delta_2[v/v_2])$ (α-renaming)

(8) $\delta[\mu v.\delta/v] = \mu v.\delta$ (folding/unfolding)

(9) $\delta[\delta_1/v] = \delta_1$ and $\delta[\delta_2/v] = \delta_2$ imply $\delta_1 = \delta_2$
if δ is contractive in v (contraction)

3.1. Type equality

The equality relation will be particularly useful for reducing any (legal) type to a canonical form.

Definition 3.1. The type *equality* relation, $\cong$ is the least congruence that satisfies the rules in Table 4.

Laws (1)–(3) state that union types are considered as equal modulo commutativity, associativity and idempotence of their components. Law (4) states that the empty type adds no information, while (5) states that the universal type makes the strongest requirements. Law (6) states that the union of two arrow types tagged with the same locality is the least upper bound with respect to the partial ordering on arrow types that will be formalized in the next section.

The remaining rules amount to stating that recursive types are equal if they denote recursive equations with equal canonical solutions. Law (7) is a generalization to union types of standard α-renaming on recursive types; indeed, $\mu v.\delta = \mu v'.\delta[v'/v]$ follows from (3) and (7). Moreover, by (7), the type $\mu v.\delta_1, \mu v.\delta_2$ is considered as equal to $\mu v.(\delta_1, \delta_2)$ in accordance with the intuition behind union types. Laws (8) and (9) are part of the standard axiomatization of equality on recursive types. Namely, (8) equates two recursive types with equal (possibly infinite) expansions. Notice that, by (8), $\mu v.\delta \cong \delta$ if v does not occur in δ. However, (8) is still too weak: we also need (9) for proving $\mu v_1.\mu v_2.\delta \cong \mu v.\delta[v/v_1, v/v_2]$. Law (9) requires δ to be *contractive* in v, i.e. v occurs in δ only under $\mapsto$.

It is well-known that equality of recursive types (rules (8) and (9) in Table 4) is decidable; a simple algorithm can be found in [12]. Table 4 simply extends equality to union types. Since unions are finite, it is easy to verify that equality remains decidable also for our types. Notice that recursive types are regular terms, namely they have finitely many subterms, since unions are finite and are equated up to commutativity, associativity and idempotence of their components.

Let us now define types in canonical form that allow us to minimize the number of components and type constructors, and to simplify case analysis when proving properties.

Definition 3.2. A type is in *canonical form* if it is generated by the following grammar:

$$\delta ::= \bot \mid \top \mid \phi_1, \ldots, \phi_n \mid \mu v.(\phi_1, \ldots, \phi_n) \quad (n \geqslant 1)$$

$$\phi ::= v \mid \ell \mapsto \pi \mapsto \delta$$

where $\phi_1, \ldots, \phi_n$ and δ satisfy the following constraints:

- all bound type variables are distinct;
- all union types $\phi_1, \ldots, \phi_k$ are such that, for all i, j ($1 \leqslant i, j \leqslant k$ and $i \neq j$), if ϕ_i and ϕ_j are variables then $\phi_i \neq \phi_j$, otherwise $L(\phi_i) \cap L(\phi_j) = \emptyset$, where $L(\delta)$, the set of *shallow localities* of δ, is defined inductively on the syntax of δ as follows:

$$L(\bot) = L(\top) = L(v) = \emptyset, \qquad L(\ell \mapsto \pi \mapsto \delta) = \{\ell\}$$

$$L(\delta_1, \delta_2) = L(\delta_1) \cup L(\delta_2), \qquad L(\mu v.\delta) = L(\delta);$$

- all μ-types $\mu v.\delta'$ are such that v occurs in δ' and it is always "guarded", i.e. on the right of an even number of $\mapsto$'s; moreover, if $\delta' = \phi_1, \ldots, \phi_n$ then $\phi_i \neq v$ for all i ($1 \leqslant i \leqslant k$).

The conditions on μ-types are quite standard, but that on union types is new. Roughly speaking, in a canonical form $\phi_1, \ldots, \phi_n$ ($n \geqslant 1$) each component ϕ_i ($1 \leqslant i \leqslant n$) is either a variable or an arrow type, with the additional constraint that all shallow localities be different. The condition about shallow localities allows us to easily determine the polarity and the type associated to a given locality within a canonical form. Notice that in a canonical recursive type $\mu v.\delta$, δ is always contractive in v.

In rewriting a type into canonical form, rules (7)–(9) are used for pushing μ-binders outside as far as possible. Let us see some examples of useful equalities on types.

- $\ell \mapsto \pi \mapsto (\ell \mapsto \pi \mapsto \nu\,[\mu\,\nu.\ell \mapsto \pi \mapsto (\ell \mapsto \pi \mapsto \nu)/\nu]) \cong \mu\nu.\ell \mapsto \pi \mapsto (\ell \mapsto \pi \mapsto \nu)$, and $\ell \mapsto \pi \mapsto (\ell \mapsto \pi \mapsto \nu[\mu\nu.\ell \mapsto \pi \mapsto \nu/\nu]) \cong \mu\nu.\ell \mapsto \pi \mapsto \nu$, by (8), from which, by (9), $\mu\nu.\ell \mapsto \pi \mapsto \nu \cong \mu\nu.\ell \mapsto \pi \mapsto (\ell \mapsto \pi \mapsto \nu)$.
- By (8), $\mu\nu.(\delta,\nu) \cong (\delta,\nu)[\mu\nu.(\delta,\nu)/\nu]$ and $(\delta,\nu)[\mu\nu.\delta/\nu] \cong \delta[\mu\nu.\delta/\nu], \mu\nu.\delta \cong \mu\nu.\delta$; then by contraction (9), $\mu\nu.(\delta,\nu) \cong \mu\nu.\delta$.
- By (8), $\ell_1 \mapsto \pi_1 \mapsto \mu\nu.(\ell_1 \mapsto \pi_1 \mapsto \nu, \ell_2 \mapsto \pi_2 \mapsto \nu), \ell_2 \mapsto \pi_2 \mapsto \mu\nu.(\ell_1 \mapsto \pi_1 \mapsto \nu, \ell_2 \mapsto \pi_2 \mapsto \nu) \cong \mu\nu.(\ell_1 \mapsto \pi_1 \mapsto \nu, \ell_2 \mapsto \pi_2 \mapsto \nu)$. Note that both are solutions of the equation $\nu = \ell_1 \mapsto \pi_1 \mapsto \nu, \ell_2 \mapsto \pi_2 \mapsto \nu$.
- By (7) and (6), $\mu\nu.\ell \mapsto \pi_1 \cup \pi_2 \mapsto \nu \cong \mu\nu_1.\ell \mapsto \pi_1 \mapsto \nu_1, \mu\nu_2.\ell \mapsto \pi_2 \mapsto \nu_2$.
- By (8) and (7), if ν does not occur in δ then $\delta, \mu\nu.\delta' \cong \mu\nu.(\delta,\delta')$.

By relying on decidability of $\cong$, we are able to reduce types to a canonical form. The actual proof, that also gives an algorithm for transforming types to canonical forms, can be found in Section 8.

Proposition 3.3. *For any legal type δ there is a canonical form δ' such that $\delta \cong \delta'$.*

For instance, if δ_1, δ_2 and δ_3 are canonical forms, then

$$\mu\nu.(\ell \mapsto \pi_1 \cup \pi_2 \mapsto (\delta_1[\nu/\nu_1], \delta_2[\nu/\nu_2]), \ell_3 \mapsto \pi_3 \mapsto \delta_3[\nu/\nu_3])$$

is a canonical form of the type $\mu\nu_1.\ell \mapsto \pi_1 \mapsto \delta_1, \mu\nu_2.\ell \mapsto \pi_2 \mapsto \delta_2, \mu\nu_3.\ell_3 \mapsto \pi_3 \mapsto \delta_3$.

From now onwards we will only consider types in canonical forms (and their unfoldings). With abuse of notation, we will continue using δ also to refer to canonical forms.

3.2. Subtyping

This section introduces the subtyping relation $\preccurlyeq$. If a process P has type δ and $\delta \preccurlyeq \delta'$ then P could be thought of as a process of type δ' too; any greater type can be assigned to a typable process simply by weakening information about its actual intentions. This is the natural intuition underlying any type-inference system with explicit subsumption. Indeed, according to our definition of subtyping (Table 6), if P_1 has type δ_1, P_2 has type δ_2 and $\delta_1 \preccurlyeq \delta_2$ then P_2 intends to perform more operations than P_1. The subtyping relation between δ_1 and δ_2 will allow us to say that P_1 can be safely used in place of P_2 but not the vice versa.

To define the subtyping relation, $\preccurlyeq$, we start by introducing an ordering between polarities, namely a hierarchy over access rights. The chosen ordering relies on the following assumptions:

- a process that can perform an **in** operation is also able to perform a **read**;
- the ability of performing **newloc** operations does not depend on the locality variables used;
- a process with polarity π posses also polarity π', for any $\pi' \subseteq \pi$.

Table 5
Subpolarity rules

$$\{i\} \sqsubseteq_{\Pi} \{r\} \quad \{n_{U_1}\} \sqsubseteq_{\Pi} \{n_{U_2}\} \quad \frac{\pi_1 \subseteq \pi_2}{\pi_2 \sqsubseteq_{\Pi} \pi_1} \quad \frac{\pi_1 \sqsubseteq_{\Pi} \pi_1' \quad \pi_2 \sqsubseteq_{\Pi} \pi_2'}{(\pi_1 \cup \pi_2) \sqsubseteq_{\Pi} (\pi_1' \cup \pi_2')}$$

Table 6
Subtyping rules

$$\bot \preccurlyeq \delta \qquad (\mathrm{ax}\bot)$$

$$\delta \preccurlyeq \top \qquad (\mathrm{ax}\top)$$

$$\frac{\delta \cong \delta'}{\delta \preccurlyeq \delta'} \qquad (\mathrm{eq})$$

$$\frac{\delta_i \preccurlyeq \delta_1' \quad \text{or} \quad \ldots \quad \text{or} \quad \delta_i \preccurlyeq \delta_k' \text{ for all } 1 \leqslant i \leqslant n}{\delta_1, \ldots, \delta_n \preccurlyeq \delta_1', \ldots, \delta_k'} \qquad (\mathrm{u/u})$$

$$\frac{\pi_2 \sqsubseteq_{\Pi} \pi_1, \quad \delta_1 \preccurlyeq \delta_2}{\ell \mapsto \pi_1 \mapsto \delta_1 \preccurlyeq \ell \mapsto \pi_2 \mapsto \delta_2} \qquad (\mapsto/\mapsto)$$

$$\frac{\ell \mapsto \pi \mapsto \delta \preccurlyeq \delta'[\mu v.\delta'/v]}{\ell \mapsto \pi \mapsto \delta \preccurlyeq \mu v.\delta'} \qquad (\mapsto/\mu)$$

$$\frac{\delta'[\mu v.\delta'/v] \preccurlyeq \ell \mapsto \pi \mapsto \delta}{\mu v.\delta' \preccurlyeq \ell \mapsto \pi \mapsto \delta} \qquad (\mu/\mapsto)$$

$$\frac{\delta_1[v/v_1] \preccurlyeq \delta_2[v/v_2]}{\mu v_1.\delta_1 \preccurlyeq \mu v_2.\delta_2} \quad \text{where } v \text{ is a fresh variable} \qquad (\mu/\mu)$$

Definition 3.4. The *subpolarity relation*, $\sqsubseteq_{\Pi}$ is the least reflexive and transitive relation closed under the rules in Table 5.

It is immediate to see that $\sqsubseteq_{\Pi}$ is decidable (it has finite domain). Notice that polarities only take into account the capabilities of processes associated to given localities, while types consider the intentions of processes, namely they also give an account of the capabilities after migration. The subtype relation below introduces a hierarchy over intentions.

Definition 3.5. The *subtyping* relation over canonical forms and unfoldings of canonical forms, $\preccurlyeq$, is the least relation closed under the rules in Table 6.

We now comment on the axioms and rules in Table 6. The empty type $\bot$ is smaller than any other type, while the universal type $\top$ is greater than any other type.

Axiom (eq) only means that if two types δ_1 and δ_2 are equated by $\cong$, then obviously $\delta_1 \preccurlyeq \delta_2 \preccurlyeq \delta_1$. In particular, since we only consider canonical forms and unfoldings of canonical forms, this rule amounts to saying that canonical forms are considered as equal up to folding/unfolding.

The type $\delta_1, \ldots, \delta_n$ represents the union, i.e. the least upper bound of types $\delta_1, \ldots, \delta_n$, where the union type constructor is monotonic and any component of the union type is smaller than the union type itself. As a consequence, $\delta_1, \ldots, \delta_n \preccurlyeq \delta'_1, \ldots, \delta'_k$ if for each δ_i there exists δ'_j such that $\delta_i \preccurlyeq \delta'_j$.

If we compare two arrow types, we have $\ell_1 \mapsto \pi_1 \mapsto \delta_1 \preccurlyeq \ell_2 \mapsto \pi_2 \mapsto \delta_2$ if, and only if, $\ell_1 = \ell_2$, $\pi_2 \sqsubseteq_{\Pi} \pi_1$ and $\delta_1 \preccurlyeq \delta_2$. This is the standard rule for subtyping over arrow types, i.e. the $\mapsto$ constructor is contravariant in its domain (i.e. polarities) and covariant in its codomain (i.e. types). Two arrow types are incomparable if $\ell_1 \neq \ell_2$.

The first two rules for recursive types permit unfolding. Since unfoldings of canonical forms are not, in general, in canonical form we have to consider these kind of types explicitly. The last rule is a simplified version of the classical rule

$$\frac{\Gamma, v_1 \preccurlyeq v_2 \vdash \delta_1 \preccurlyeq \delta_2}{\Gamma \vdash \mu v_1.\delta_1 \preccurlyeq \mu v_2.\delta_2}$$

where Γ is a binding context for type variables and subtyping assumptions, and v_1 (v_2) occurs only in δ_1 (δ_2). The rule above takes into account that recursion variables do not necessarily occur in positive positions. However, since we only consider positive types, hence monotonic in the recursion variable, a simpler rule can be used, i.e. if $\delta_1 \preccurlyeq \delta_2$ then $\mu v.\delta_1 \preccurlyeq \mu v.\delta_2$. This allows us to avoid contexts with subtyping assumptions in the derivations. Rule (μ/μ) in Table 6 uses this property together with the fact that types are equal up to renaming of bound variables.

One can prove that transitivity is an admissible rule for the subtyping relation (the actual proof is postponed to Section 8). As a consequence, since types are positive and type constructors are monotonic, we have that types are monotonic by type substitution.

Proposition 3.6. *Let δ', δ'' and δ''' be canonical forms. If $\delta' \preccurlyeq \delta''$ and $\delta'' \preccurlyeq \delta'''$ are provable, then $\delta' \preccurlyeq \delta'''$ is provable.*

Corollary 3.7. *Let δ_1, δ_2, δ' and δ'' be canonical forms. If $\delta_1 \preccurlyeq \delta_2$ and $\delta' \preccurlyeq \delta''$ then $\delta_1[\delta'/v] \preccurlyeq \delta_2[\delta''/v]$.*

Decidability of subtyping can now be established. The rules in Table 6 are syntax-driven. Any proof of $\delta_1 \preccurlyeq \delta_2$ can be constructed by a simple two-step algorithm: first try to prove $\delta_1 \cong \delta_2$ and if you fail then apply the rules backwards in a syntax-directed fashion. Since $\cong$ is decidable, the algorithm for establishing $\delta_1 \preccurlyeq \delta_2$ is completely determined once we establish that the axioms (ax$\bot$), (ax$\top$) and (eq) have priority over the other rules. The proof of decidability, that explicitly defines the algorithm, can be found in Section 8.

Theorem 3.8. *The subtyping relation $\preccurlyeq$ is decidable.*

A more general discussion about equality and subtyping of recursive types can be found in [5]. Moreover, an efficient algorithm for deciding subtyping in the presence of recursive types has been proposed in [27]. In our context, decidability of subtyping involves both union types and recursive types. However, in any type, and then in its canonical forms, recursive types can only occur in positive positions and satisfy the constraints in Definition 3.2, thus restricting the problem of subtyping decidability.

4. A capability-based type system

This section introduces the type inference system and the notion of well-typed net. The inference system is used in the first phase of the typing analysis for assigning types to processes.

4.1. The type inference system

Type contexts, Γ are functions mapping process variables and identifiers into types. They are written as sequences of type assignments, i.e. $X_1 : \delta_1, \ldots, X_n : \delta_n$. The symbol ϕ denotes the empty context.

Given a type context Γ, we write $\Gamma[\delta/X]$ to denote either the extension of Γ with the assignment $X : \delta$ (when X is unbound in Γ), or the updating of Γ that binds X to δ. The auxiliary function *update*, defined structurally over tuples syntax, behaves like the identity function for all fields but $!X : \delta$. Formally, it is defined by

$$update(\Gamma, t) = \begin{cases} update(update(\Gamma, f), t') & \text{if } t = f, t' \\ \Gamma[\delta/X] & \text{if } t = !X : \delta \\ \Gamma & \text{otherwise} \end{cases}$$

The type judgments for processes take the form $\Gamma \vdash P : \delta$ where Γ is a type context providing the type of process variables and identifiers in P. A statement such as $\Gamma \vdash P : \delta$ asserts that, within the context Γ, the intentions of P are those specified by δ.

In the conclusion of the inference rules, we will write δ_1, δ_2 to denote the canonical form of the type δ_1, δ_2 obtained by applying the rules in Table 4. For instance, we write that from $\Gamma \vdash P : \ell \mapsto \pi \mapsto \delta$ we derive $\Gamma \vdash \mathbf{out}(t)@\ell.P : \ell \mapsto \pi \mapsto \delta, \ell \mapsto \{o\} \mapsto \bot$ but actually we mean $\Gamma \vdash \mathbf{out}(t)@\ell.P : \ell \mapsto \pi \cup \{o\} \mapsto \delta$.

Notation 4.1. *Let δ be a canonical form. The following notations are inductively defined*:

- $\delta \downarrow_p \ell$:
  - $\bot \downarrow_p \ell = \top \downarrow_p \ell = v \downarrow_p \ell = \emptyset$;
  - $(\ell' \mapsto \pi \mapsto \delta') \downarrow_p \ell = \begin{cases} \emptyset & \text{if } \ell \neq \ell' \\ \pi & \text{otherwise;} \end{cases}$
  - $(\delta_1, \ldots, \delta_n) \downarrow_p \ell = \delta_1 \downarrow_p \ell \cup \cdots \cup \delta_n \downarrow_p \ell$;
  - $(\mu v.\delta') \downarrow_p \ell = \delta' \downarrow_p \ell$.

- $\delta\downarrow_t\ell$:
  - $\bot\downarrow_t\ell=\top\downarrow_t\ell=v\downarrow_t\ell=\bot$;
  - $(\ell'\mapsto\pi\mapsto\delta')\downarrow_t\ell=\begin{cases}\bot & \text{if } \ell\neq\ell'\\ \delta' & \text{otherwise};\end{cases}$
  - $(\delta_1,\ldots,\delta_n)\downarrow_t\ell=\delta_1\downarrow_t\ell,\ldots,\delta_n\downarrow_t\ell$;
  - $(\mu v.\delta')\downarrow_t\ell=\delta'[\mu v.\delta'/v]\downarrow_t\ell$.
- $\delta\Downarrow_t\ell$:
  - $\bot\Downarrow_t\ell=\top\Downarrow_t\ell=v\Downarrow_t\ell=\bot$;
  - $(\ell'\mapsto\pi\mapsto\delta')\Downarrow_t\ell=\begin{cases}\delta'\Downarrow_t\ell & \text{if } \ell\neq\ell'\\ \delta' & \text{otherwise};\end{cases}$
  - $(\delta_1,\ldots,\delta_n)\Downarrow_t\ell=\delta_1\Downarrow_t\ell,\ldots,\delta_n\Downarrow_t\ell$;
  - $(\mu v.\delta')\Downarrow_t\ell=(\delta'[v'/v]\Downarrow_t\ell)[\mu v.\delta'/v']$, *where* v' *is fresh.*
- $\delta\{\delta'/\!/\ell\}$:
  - $\bot\{\delta'/\!/\ell\}=\bot,\ \top\{\delta'/\!/\ell\}=\top,\ v\{\delta'/\!/\ell\}=\bot$;
  - $(\ell'\mapsto\pi\mapsto\delta'')\{\delta'/\!/\ell\}=\begin{cases}\ell'\mapsto\pi\mapsto\delta''\{\delta'/\!/\ell\} & \text{if } \ell\neq\ell'\\ \ell\mapsto\pi\mapsto\delta' & \text{otherwise};\end{cases}$
  - $(\delta_1,\ldots,\delta_n)\{\delta'/\!/\ell\}=\delta_1\{\delta'/\!/\ell\},\ldots,\delta_n\{\delta'/\!/\ell\}$;
  - $(\mu v.\delta'')\{\delta'/\!/\ell\}=\mu v'.(\delta''[v'/v]\{\delta'/\!/\ell\})$, *where* v' *is fresh.*
- $\delta\{\rho\}$:
  - $\bot\{\rho\}=\bot,\ \top\{\rho\}=\top,\ v\{\rho\}=\bot$;
  - $(\ell\mapsto\pi\mapsto\delta')\{\rho\}=\ell\{\rho\}\mapsto\pi\mapsto\delta'$;
  - $(\delta_1,\ldots,\delta_n)\{\rho\}=\delta_1\{\rho\},\ldots,\delta_n\{\rho\}$;
  - $(\mu v.\delta')\{\rho\}=\mu v.(\delta'\{\rho\})$.

Let us comment on the notations introduced above. $\delta\downarrow_p\ell$ and $\delta\downarrow_t\ell$ are used to denote the polarity and the type associated to ℓ in δ, respectively. Type $\delta\downarrow_t\ell$ does not take into account the types relative to those occurrences of ℓ within δ that are on the right of $\mapsto$ constructors. We use $\delta\Downarrow_t\ell$ to denote the union of all the types associated to the outermost occurrences of ℓ within δ. Notice that when $\Downarrow_t$ is applied to a recursive type, we pick up the union of the types associated to the outermost occurrences of ℓ in the body of the recursive type. The unfolding takes place only after this type has been obtained; this ensures that $\delta\Downarrow_t\ell$ is well-defined. $\delta\{\delta'/\!/\ell\}$ is used to denote the type obtained from δ by replacing with δ' the types related to the outermost occurrences of ℓ. Type $\delta\{\rho\}$ is obtained from δ by using ρ to interpret shallow localities of δ.

The rules of the type inference system are given in Table 7. We will say that a process term P is *typable* if there exist Γ and δ such that $\Gamma\vdash P:\delta$ is provable by using the rules in Table 7.

The type of a process variable (or identifier) is completely determined by the type context, Γ. Definedness of $\Gamma(X)$ is guaranteed by the fact that processes are closed terms. The simplest process (the null process **nil**) has no intentions at all.

Table 7
KLAIM type inference rules (we assume that $A(\tilde{X} : \widetilde{\delta_X}, \tilde{u} : \widetilde{\delta_u}, \tilde{x}) \stackrel{\text{def}}{=} P$)

$$\Gamma \vdash X : \Gamma(X) \qquad \Gamma \vdash A : \Gamma(A) \tag{1}$$

$$\Gamma \vdash \mathbf{nil} : \bot \tag{2}$$

$$\frac{\Gamma \vdash P : \delta}{\Gamma \vdash \mathbf{out}(t)@\ell.P : (\delta, \ell \mapsto \{o\} \mapsto \bot)} \tag{3}$$

$$\frac{update(\Gamma, t) \vdash P : \delta \quad \delta \Downarrow_t u \preccurlyeq \delta_u \text{ for all } (!\, u : \delta_u) \in \{t\}}{\Gamma \vdash \mathbf{read}(t)@\ell.P : (\delta, \ell \mapsto \{r\} \mapsto \bot)} \tag{4}$$

$$\frac{update(\Gamma, t) \vdash P : \delta \quad \delta \Downarrow_t u \preccurlyeq \delta_u \text{ for all } (!\, u : \delta_u) \in \{t\}}{\Gamma \vdash \mathbf{in}(t)@\ell.P : (\delta, \ell \mapsto \{i\} \mapsto \bot)} \tag{5}$$

$$\frac{\Gamma \vdash P : \delta \quad \Gamma \vdash Q : \delta'}{\Gamma \vdash \mathbf{eval}(Q)@\ell.P : (\delta, \ell \mapsto \{e\} \mapsto \delta')} \tag{6}$$

$$\frac{\Gamma \vdash P : \delta \quad \delta \Downarrow_t u \preccurlyeq \delta'}{\Gamma \vdash \mathbf{newloc}(u : \delta').P : (\delta\{\delta' /\!/ u\}, \mathbf{self} \mapsto \{n_{\{u\}}\} \mapsto \bot)} \tag{7}$$

$$\frac{\Gamma \vdash P : \delta_1 \quad \Gamma \vdash Q : \delta_2}{\Gamma \vdash P \mid Q : (\delta_1, \delta_2)} \tag{8}$$

$$\frac{\Gamma[\widetilde{\delta_X}/\tilde{X}][\delta/A] \vdash P : \delta' \quad \delta \cong \delta' \quad \delta \Downarrow_t u_i \preccurlyeq \delta_{u_i} \text{ for all } u_i \in \{\tilde{u}\}}{\Gamma \vdash A : \delta} \tag{9}$$

$$\frac{\Gamma \vdash A : \delta \quad \Gamma \vdash P_i : \delta_i \text{ and } \delta_i \preccurlyeq \delta_{X_i} \text{ for all } P_i \in \{\tilde{P}\}}{\Gamma \vdash A\langle \tilde{P}, \tilde{\ell}, \vec{e} \rangle : \delta[\tilde{\ell}/\tilde{u}]} \tag{10}$$

The typing rule for **out** states that the type of $\mathbf{out}(t)@\ell.P$ extends the capabilities of P at ℓ with o. Since **out** is not a binder, P is typed within the same context as $\mathbf{out}(t)@\ell.P$.

The typing rules for **read** and **in** extend the type of P at ℓ with the corresponding capability (r or i). The type of P is derived in a context updated with the type assignments for the process variables bound by **read** and **in**. The rules apply only whenever process P properly use the locality variables bound by **read** and **in**. This amounts to saying that, for each locality variable u with type δ_u, the remote operations of P at u ($\delta \Downarrow_t u$) do respect δ_u. The typing rule for **eval** extends the type of P at ℓ with e and records that the operations of P at ℓ have to be extended with those (δ') of the spawned process Q.

The typing rule for **newloc** extends the type of P at **self** with $n_{\{u\}}$ and at u with type δ'; moreover, it checks whether the operations that P is willing to perform at u ($\delta \Downarrow_t u$) comply with δ'. Notice that the inferred type contains information for mapping locality variable u to the site where the **newloc** operation is performed when checking well-typedness of a net (see Definition 4.9).

The typing rule for parallel composition states that the intentions of the composed process are the union of those of the components, while the binding context is left unchanged. We would like to remark that also in [20] union types have been used for typing parallel processes.

The typing rule for (possibly recursive) process definition (rule (9)), updates the type context with the types declared for the process variables occurring as parameters in the definition and with the binding between the process identifier A and a (possibly recursive) candidate type δ. The resulting context is exploited to infer (up to type equality) the type δ for P. Similarly to the typing rules for **read** and **in**, for each formal locality variable u_i, one checks whether the operations of P at u_i (i.e. $\delta \Downarrow_t u_i$) match the type declaration δ_{u_i}. Finally, the inferred type is δ.

The last typing rule is the rule for process invocation. First, it determines the type of the process identifier and those of the process arguments. Then, it checks whether the type inferred for any process argument agrees with that of the corresponding formal parameter. No requirement is imposed on the other arguments. Types of locality variables are controlled when one of the rules for **in**, **read** and **newloc** is applied. Types of localities are controlled when well-typedness of nets is checked. The inferred type states that $A\langle \tilde{P}, \tilde{\ell}, \vec{e}\rangle$ intends to perform at $\tilde{\ell}$ the same operations of $A\langle \tilde{X}, \tilde{u}, \tilde{x}\rangle$ at $\tilde{u}$. Soundness of the application of $[\tilde{l}/\tilde{u}]$ to δ follows from the assumption that all bound names in the definition of A are distinct.

Remark 4.2. Subtyping is crucial in typing KLAIM processes. Most of the rules in Table 7, after recording in the type all the operations the process intends to perform, check this type with subtyping constraints. A type δ matches these constraints if, and only if, either δ or a greater δ' do so. From a semantical point of view, premises of the form $\delta \preccurlyeq \delta'$ play the same role of the subsumption rule

$$\frac{\Gamma \vdash P : \delta \quad \delta \preccurlyeq \delta'}{\Gamma \vdash P : \delta'}.$$

This rule has not been explicitly introduced because with it we would have lost the guarantee that our type system derives a unique type for any typable non recursive process.

We now prove the following *substitution property* (for possibly open process terms).

Lemma 4.3. *Let $\Gamma[\delta_X/X] \vdash P : \delta$. If $\Gamma \vdash Q : \delta_Q$ and $\delta_Q \preccurlyeq \delta_X$, then $\Gamma \vdash P[Q/X] : \delta'$ for some $\delta' \preccurlyeq \delta$.*

Proof. By induction on the length of the derivation of $\Gamma[\delta_X/X] \vdash P : \delta$. By case analysis on the last inference rule applied, it can be easily verified that δ_X may occur in δ only in positive positions, and then, by Corollary 3.7, it can be concluded that $\delta_Q \preccurlyeq \delta_X$ implies $\delta' \preccurlyeq \delta$. □

Corollary 4.4. *Let $P_1 = P_2[\tilde{P}/\tilde{X}, \tilde{l}/\tilde{u}, \tilde{v}/\tilde{x}]$ and suppose that $\Gamma[\widetilde{\delta_X}/\tilde{X}] \vdash P_2 : \delta$. If $\Gamma \vdash P_i : \delta_{P_i}$ and $\delta_{P_i} \preccurlyeq \delta_{X_i}$ for any $P_i \in \{\tilde{P}\}$, then $\Gamma \vdash P_1 : \delta'$ for some $\delta' \preccurlyeq \delta[\tilde{l}/\tilde{u}]$.*

We now establish decidability of the type inference system. It is easy to verify that for any process P such that $\Gamma \vdash P : \delta$ may be derived, the following properties hold:

1. if $P = \mathbf{nil}$ then $\delta = \bot$;
2. if $P = X$ then $\delta = \Gamma(X)$;
3. if $P = a.Q$ then δ is completely determined by the type derived for Q in the context obtained by updating Γ with the types of all the process variables bound by a and, in case $a = \mathbf{eval}(R)@\ell$, by the type derived for R in Γ;
4. if $P = A\langle \tilde{P}, \tilde{l}, \tilde{v} \rangle$, where $A(\tilde{X} : \tilde{\delta}, \tilde{u} : \tilde{\delta}, \tilde{x}) \stackrel{\text{def}}{=} P$, then δ is completely determined by the types of A, $\tilde{P}$ and $\tilde{X}$, according to rule (10);
5. if $P = Q_1 \mid Q_2$ then δ is the union of the types of Q_1 and Q_2.

These properties tell us that the type inference system without rule (9) is monomorphic, i.e. only one type can be deduced in a context Γ for a typable process. The deduced type can be constructed from the syntax of the process by using the types of its subterms. In the case of rule (9) the type inference system behaves differently. When $A(\tilde{X} : \tilde{\delta}, \tilde{u} : \tilde{\delta}, \tilde{x}) \stackrel{\text{def}}{=} P$, if A is typable, several types can be assigned to it using rule (9). However, a *minimal type* exists such that all the deducible types are greater than it. The proof of this property, that also contains an algorithm to compute the minimal type, can be found in Section 8.

Theorem 4.5. *If $\Gamma \vdash P : \delta'$ then there exists a minimal type δ such that $\Gamma \vdash P : \delta$ and $\delta \preccurlyeq \delta''$ for all δ'' such that $\Gamma \vdash P : \delta''$.*

Corollary 4.6. *For any typable process P, a minimal type δ exists such that*

(i) $\phi \vdash P : \delta$,
(ii) $\delta \preccurlyeq \delta'$ *for any* δ' *such that* $\phi \vdash P : \delta'$,
(iii) $\phi \vdash P : \delta$ *is decidable.*

Proof. (i) and (ii) follow from Theorem 4.5 and from closedness of P, (iii) follows from (i) and from decidability of $\preccurlyeq$. □

The main impact of the existence of a minimal type is that the type inference system is decidable.

Corollary 4.7. *For any process P, the existence of a type δ such that $\phi \vdash P : \delta$ is decidable.*

4.2. Well-typed nets

This section introduces the notion of well-typed net. For a net to be well-typed, it will be required that the types of the processes in the net agree with the access rights of the sites where they are located. More specifically, the types of the processes, as determined by the type inference system, are checked against those fixed by the net coordinator, while taking into account where each process has been located.

To compare process types as inferred from the inference system with the types of the nodes of the net where processes are allocated, localities have to be mapped into sites by using site allocation environments, i.e. localities must be *interpreted*.

Definition 4.8. The *type interpretation function* associated to a net N_S, $\Theta_{N_S} : S \to \mathscr{E}$ is defined as follows: for all $s \in S$, $\Theta_{N_S}(s) = \rho_s$ if $s ::_{\rho_s}^{\delta_s} P \in N_S$, for some δ_s and P.

The type interpretation function is well-defined because N_S enjoys conditions 1. and 2. of Definition 2.2. We will write $\Theta_{N_S}[\rho/s]$ to denote the extension of $\Theta_{N_S} : S \to \mathscr{E}$ to $s \notin S$ that yields ρ.

Definition 4.9. Given a type interpretation function Θ_{N_S}, an $s \in S$ and a type δ, the *interpretation* $\delta\{\Theta_{N_S}\}_s$ of δ in s is (a canonical form of) the type without occurrences of locality variables defined inductively on the syntax of δ as follows:

- $\bot\{\Theta_{N_S}\}_s = \bot$, $\top\{\Theta_{N_S}\}_s = \top$, $v\{\Theta_{N_S}\}_s = v$;
- $(\delta_1, \ell \mapsto \{n_U\} \cup \pi \mapsto \delta_2)\{\Theta_{N_S}\}_s = (\delta_1[s/U]\{\Theta_{N_S}\}_s)$, $((\ell \mapsto \{n\} \cup \pi \mapsto \delta_2[s/U])\{\Theta_{N_S}\}_s)$;
- if there is no U such that $n_U \in \pi$, then

$$(\ell \mapsto \pi \mapsto \delta')\{\Theta_{N_S}\}_s = \begin{cases} \ell\{\rho_s\} \mapsto \pi \mapsto \delta'\{\Theta_{N_S}\}_{\ell\{\rho_s\}} & \text{if } \ell\{\rho_s\} \in S \\ \bot & \text{if } \ell \in TVloc \\ \ell \mapsto \pi \mapsto \delta' & \text{otherwise;} \end{cases}$$

- $(\delta_1, \delta_2)\{\Theta_{N_S}\}_s = \delta_1\{\Theta_{N_S}\}_s, \delta_2\{\Theta_{N_S}\}_s$;
- $(\mu v.\delta')\{\Theta_{N_S}\}_s = \mu v.\delta'\{\Theta_{N_S}\}_s$.

where $[s/U]$ denotes the substitution $[s/u_1, \ldots, s/u_n]$ if $U = \{u_1, \ldots, u_n\}$.

When inferring types of processes, the locality variables used to denote newly created sites are interpreted just as local sites. The index of capability n serves this purpose: it permits to recover the bindings between locality variables and the site where they are bound. While interpreting an arrow type, to correctly replace all locality variables in δ, when an n_U is encountered, a substitution function is defined that is also applied to the context of the arrow type. When interpreting arrow types whose polarities do not contain an indexed n, three different subcases, depending on the interpretation of the locality on the left of $\mapsto$, have to be taken into account:

- either the locality can be interpreted as a site of the net,
- or the locality is a variable for inputting sites,
- or none of the two previous cases arises (e.g. it is a locality that cannot be mapped to a site of the net).

Locality variables used in tuples for inputting sites do not give any contribution to the interpreted type because the intentions of processes relative to them are statically checked during the type inference phase. For this reason, the resulting type in the second item above is $\bot$. The last item above takes into account localities that cannot be mapped to sites of the net; in this case the type is left unchanged.

Table 8
Tuple evaluation function

$\mathscr{T}[\![e]\!]_\rho = \mathscr{E}[\![e]\!]$	$\mathscr{T}[\![!\,x]\!]_\rho = !\,x$
$\mathscr{T}[\![P]\!]_\rho = P\{\rho\}$	$\mathscr{T}[\![!\,X:\delta]\!]_\rho = !\,X:\delta$
$\mathscr{T}[\![\ell]\!]_\rho = \rho(\ell)$	$\mathscr{T}[\![!\,u:\delta]\!]_\rho = !\,u:\delta$
$\mathscr{T}[\![f,t]\!]_\rho = \mathscr{T}[\![f]\!]_\rho, \mathscr{T}[\![t]\!]_\rho$	

Definition 4.10. A net N_S is *well-typed* if for any node $s ::^{\delta_s}_{\rho_s} P$, P is typable and $\phi \vdash P : \delta$, with δ minimal type for P, implies $\delta\{\Theta_{N_S}\}_s \preccurlyeq \delta_s$.

The well-typedness condition implies that processes cannot use a locality if this is not known in the allocation environment of the node.

5. Operational semantics

The operational semantics of KLAIM nets of processes is presented in Table 11. To give a simple presentation of the operational rules, the structural congruence, $\equiv$ defined as the least congruence relation closed under the rules in Table 10, is introduced. Hereafter, we use ℓ for denoting localities, locality variables and sites and assume that allocation environments are extended to sites; over them they act as the identity function. For further explanations and comments we refer the interested reader to [18].

Like in [19, 33], we model tuples as processes. To this aim, we extend KLAIM syntax with processes of the form **out**(et) for denoting *evaluated tuples* (referred to as et).

The evaluation function for tuples, $\mathscr{T}[\![\ ]\!]_\rho$, makes use of an allocation environment for resolving locality names and relies on an evaluation mechanism, $\mathscr{E}[\![\ ]\!]$, for closed expressions (i.e. expressions without variables). In Table 8, that inductively defines $\mathscr{T}[\![\ ]\!]_\rho$, there is only one non-trivial case, namely the evaluation of a process $\mathscr{T}[\![P]\!]_\rho$. This yields a *process closure* $P\{\rho\}$ that stands for the process P packaged with the allocation environment ρ. Note that types assigned to locality and process variables in formals are not "evaluated". They both will be evaluated by the pattern matching predicate when the site to which the locality variable must be bound and the process to which the process variable must be bound are known.

Process closures are evaluated using the least congruence induced by the laws in Table 9 that allow us to push closures inside the structure of processes. In $P\{\rho\}$, a locality ℓ used in P is interpreted as stated in ρ; however, ρ can be a "partial" environment, namely there can be localities that are left unchanged when ρ is applied because they are not in its domain. For the evaluation of $t\{\rho\}$, there are two missing cases in the laws in Table 9. The case $\ell\{\rho\}$ is already covered by the notational

Table 9
Closure laws

$$
\begin{aligned}
\mathbf{nil}\{\rho\} &= \mathbf{nil} \\
X\{\rho\} &= X \\
(\mathbf{out}(t)@\ell.P)\{\rho\} &= \mathbf{out}(t\{\rho\})@\ell\{\rho\}.P\{\rho\} \\
(\mathbf{eval}(Q)@\ell.P)\{\rho\} &= \mathbf{eval}(Q)@\ell\{\rho\}.P\{\rho\} \\
(\mathbf{in}(t)@\ell.P)\{\rho\} &= \mathbf{in}(t\{\rho\})@\ell\{\rho\}.P\{\rho\} \\
(\mathbf{read}(t)@\ell.P)\{\rho\} &= \mathbf{read}(t\{\rho\})@\ell\{\rho\}.P\{\rho\} \\
(\mathbf{new}(u:\delta).P)\{\rho\} &= \mathbf{new}(u:\delta).P\{\rho\} \\
(P_1 \mid P_2)\{\rho\} &= P_1\{\rho\} \mid P_2\{\rho\} \\
A\langle \tilde{P}, \tilde{\ell}, \tilde{e}\rangle\{\rho\} &= P[\tilde{P}/\tilde{X}, \tilde{\ell}/\tilde{u}, \tilde{e}/\tilde{x}]\{\rho\} \qquad \text{if } A(\tilde{X}:\tilde{\delta}, \tilde{u}:\tilde{\delta}, \tilde{x}) \stackrel{\text{def}}{=} P \\
\\
e\{\rho\} &= e \\
!x\{\rho\} &= !x \\
(!X:\delta)\{\rho\} &= !X:\delta\{\rho\} \\
(!u:\delta)\{\rho\} &= !u:\delta \\
(f,t)\{\rho\} &= f\{\rho\}, t\{\rho\}
\end{aligned}
$$

convention introduced in Section 2.3. The case $P\{\rho\}$ is covered by the laws relative to process closures.

The first two structural laws in Table 10 say that $\|$ is commutative and associative. The third law permits recollecting (spawning) the null process. The last law relies on the fact that the environments of sites cannot be dynamically modified; hence, it is always possible to distribute (recover) the processes located onto the same node over (from) clones of that node.

We now introduce the notion of *conservative extension* of the function used to induce the types of the nodes of a net. This notion is used to derive the new types of the nodes of the net in case of dynamic reconfiguration. Indeed, when a new node is created, the types of the nodes of the net have to be "extended" by adding the rights of the existing nodes over the new one and the rights of the new node over the existing ones. For obtaining the type of the new node, first the type assigned to the locality variable (the argument of **newloc**) is interpreted by using the local allocation environment. Then, the interpreted type is used to derive the access rights of the new node with respect to the other nodes. Finally, the new types of the nodes of the net are obtained as the solution of a new set of type equations. An extension is called

Table 10
Congruence laws

$$N_1 \parallel N_2 = N_2 \parallel N_1$$
$$(N_1 \parallel N_2) \parallel N_3 = N_2 \parallel (N_1 \parallel N_3)$$
$$s ::_\rho^\delta P = s ::_\rho^\delta (P \mid \mathbf{nil})$$
$$s ::_\rho^\delta (P_1 \mid P_2) = s ::_\rho^\delta P_1 \parallel s ::_\rho^\delta P_2$$

"conservative" whenever the type specified by the programmer for the new node is compatible with those induced by the extension.

Definition 5.1. Let S be a finite set of sites and $\Lambda : S \rightharpoonup (S \rightharpoonup \Pi)$ be a partial function. The *extension*, $\Lambda' : (S \cup \{s'\}) \rightharpoonup ((S \cup \{s'\}) \rightharpoonup \Pi)$ of Λ with respect to $s' \notin S$, δ, u, a type interpretation function Θ with $dom(\Theta) = S \cup \{s'\}$ and $s \in S$, is defined as follows:
- for all $s_1, s_2 \in S$, $\Lambda'(s_1)(s_2) = \Lambda(s_1)(s_2)$,
- for all $s_1 \in S$, $\Lambda'(s_1)(s') = \Lambda(s_1)(s)$, $\Lambda'(s')(s_1) = \delta[s'/u]\{\Theta(s')\} \downarrow_p s_1$ and $\Lambda'(s')(s') = \delta[s'/u]\{\Theta(s')\} \downarrow_p s'$,

whenever the right-hand sides of the above equalities differ from the empty set. Λ' is a *conservative* extension, if the following conditions hold:
1. if $\Lambda'(s)(s)$ is defined then $\Lambda'(s)(s) \sqsubseteq_\Pi \Lambda'(s')(s')$,
2. for all $s_1 \in S$ such that $\Lambda'(s)(s_1)$ is defined we have $\Lambda'(s)(s_1) \sqsubseteq_\Pi \Lambda'(s')(s_1)$,
3. $s_j \mapsto \pi \mapsto \delta' \in Sub(\delta[s'/u]\{\Theta(s')\}_{s'})$ and $\delta' \downarrow_p s_1 \neq \emptyset$, where $s_1 \in dom(\Theta)$, imply $\Lambda'(s_j)(s_1) \sqsubseteq_\Pi \delta' \downarrow_p s_1$.

Notation 5.2. *In Table* 11 *the following shorthands are adopted*:
- δ_P *stands for a type equivalent to* $\delta'_P[\tilde{\ell}/\tilde{\mu}]$, *where* δ'_P *is such that* $update(\phi, t) \vdash P : \delta'_P$ *and* $[\tilde{\ell}/\tilde{\mu}]$ *is obtained by restricting* $[et/\mathscr{T}[\![t]\!]_\rho]$ *to locality variables*;
- $\Theta_{N_{S'}}$ *is the type interpretation function associated to* $N_{S'} : \Lambda$, *the net before reduction*;
- $\delta^*_{s_i}$, *for* $s_i \in S \cup \{s, s'\}$, *are the components of the solution of the system of type equations induced by* Λ', *the conservative extension of* Λ *with respect to* s', δ, u, $\Theta_{N_{S'}}[\rho_{s'}/s']$ *and* s;
- $N_S[\delta^*_{s_i}/\delta_{s_i}]_{s_i \in S}$ *is the net* N_S *where the types* δ_{s_i} *of the nodes are replaced by* $\delta^*_{s_i}$.

Rule (1) in Table 11 says that the execution of an *out* operation adds a tuple to a tuple space. The local allocation environment is used both for determining the site where the tuple must be placed and for evaluating the tuple. In particular, if the tuple contains a field with a process, the corresponding field of the evaluated tuple contains the process resulting from the evaluation of a closure. Hence, processes in a tuple are transmitted after the interpretation of the used localities by using the local allocation

Table 11
KLAIM operational semantics (see Notation 5.2)

$$\frac{s' = \rho(\ell) \quad et = \mathscr{T}[\![t]\!]_\rho}{N_S \| s ::^{\delta}_{\rho} \mathbf{out}(t)@\ell \,.\, P \| s' ::^{\delta'}_{\rho'} P' \rightarrowtail N_S \| s ::^{\delta}_{\rho} P \| s' ::^{\delta'}_{\rho'} (P' \mid \mathbf{out}(et))} \quad (1)$$

$$\frac{s' = \rho(\ell)}{N_S \| s ::^{\delta}_{\rho} \mathbf{eval}(Q)@\ell \,.\, P \| s' ::^{\delta'}_{\rho'} P' \rightarrowtail N_S \| s ::^{\delta}_{\rho} P \| s' ::^{\delta'}_{\rho'} (P' \mid Q)} \quad (2)$$

$$\frac{s' = \rho(\ell) \quad match(\mathscr{T}[\![t]\!]_\rho, et, s, \delta_P, \Theta_{N_{s'}})}{N_S \| s ::^{\delta}_{\rho} \mathbf{in}(t)@\ell \,.\, P \parallel s' ::^{\delta'}_{\rho'} \mathbf{out}(et) \rightarrowtail N_S \| s ::^{\delta}_{\rho} P[et/\mathscr{T}[\![t]\!]_\rho] \parallel s' ::^{\delta'}_{\rho'} \mathbf{nil}} \quad (3)$$

$$\frac{s' = \rho(\ell) \quad match(\mathscr{T}[\![t]\!]_\rho, et, s, \delta_P, \Theta_{N_{s'}})}{N_S \| s ::^{\delta}_{\rho} \mathbf{read}(t)@\ell \,.\, P \| s' ::^{\delta'}_{\rho'} \mathbf{out}(et) \rightarrowtail N_S \| s ::^{\delta}_{\rho} P[et/\mathscr{T}[\![t]\!]_\rho] \| s' ::^{\delta'}_{\rho'} \mathbf{out}(et)} \quad (4)$$

$$\frac{s' \notin S \cup \{s\} \quad \rho_{s'} = \rho_s[s'/\mathbf{self}]}{N_S \| s ::^{\delta_s}_{\rho_s} \mathbf{newloc}(u : \delta) \,.\, P \rightarrowtail N_S[\delta^*_{s_i}/\delta_{s_i}]_{s_i \in S} \parallel s ::^{\delta^*_s}_{\rho_s} P[s'/u] \parallel s' ::^{\delta^*_{s'}}_{\rho_{s'}} \mathbf{nil}} \quad (5)$$

$$\frac{N_S \| s ::^{\delta}_{\rho} P[\tilde{P}/\tilde{X}, \tilde{\ell}/\tilde{r}, \tilde{e}/\tilde{x}] \rightarrowtail N}{N_S \| s ::^{\delta}_{\rho} A\langle \tilde{P}, \tilde{\ell}, \tilde{e} \rangle \rightarrowtail N} \quad A(\tilde{X} : \tilde{\delta}, \tilde{r} : \tilde{\delta}, \tilde{x}) \stackrel{\text{def}}{=} P \quad (6)$$

$$\frac{N \equiv N_1 \quad N_1 \rightarrowtail N_2 \quad N_2 \equiv N'}{N \rightarrowtail N'} \quad (7)$$

environment. This corresponds to having a *static scoping* discipline for the (remote) generation of tuples.

A *dynamic scoping* strategy is adopted for the **eval** operation, described by rule (2). In this case the localities of the spawned process are not interpreted by using the local allocation environment. Instead, the process is transmitted and its execution can be influenced by the remote allocation environment.

Rule (3) says that a process can perform an **in** action by synchronizing with a process which represents a matching tuple *et*. Matching (defined in Table 12) takes as arguments the two candidate tuples, the site where the operation is executed, the type of the continuation (that is statically derived and retrieved from the symbol table when it is needed) and the type interpretation function of the net. The result of this synchronization is that tuple *et* is consumed, i.e. the corresponding process becomes **nil**, and its values are used to replace, within the process which has performed the **in** operation, the free occurrences of the corresponding variables of t (this substitution is denoted by $[et/\mathscr{T}[\![t]\!]_\rho]$).

Rule (4) can be interpreted similarly to (3). Only notice that, while **in** modifies the tuple space, **read** does not; in the conclusions of rule (4) the tuple space is left unchanged by process evolution.

Rules (1)–(4) do not introduce new sites. The creation of a new node, made possible by **newloc**, is described in rule (5). The new node inherits (part of) the access rights and all the knowledge of the creating node. The environment of the new node is derived

Table 12
Matching rules

$$match(v, v, s, \delta, \Theta_{N_S}) \qquad match(P, P, s, \delta, \Theta_{N_S}) \qquad match(s, s, s', \delta, \Theta_{N_S})$$

$$match(v, !\,x, s, \delta, \Theta_{N_S}) \quad \frac{\phi \vdash P : \delta'' \quad \delta''\{\Theta_{N_S}\}_s \preccurlyeq \delta\{\Theta_{N_S}\}_s}{match(P, !\,X : \delta, s, \delta', \Theta_{N_S})} \qquad \frac{\delta[s/u]\{\Theta_{N_S}\}_s \preccurlyeq \delta_s}{match(s, !\,u : \delta, s', \delta', \Theta_{N_S})}$$

$$match(!\,x, v, s, \delta, \Theta_{N_S}) \quad \frac{\phi \vdash P : \delta'' \quad \delta''\{\Theta_{N_S}\}_s \preccurlyeq \delta\{\Theta_{N_S}\}_s}{match(!\,X : \delta, P, s, \delta', \Theta_{N_S})}$$

$$\frac{\delta[s/u]\{\Theta_{N_S}\}_s \preccurlyeq \delta_s \quad \Pi_s(u, s', \delta', \Theta_{N_S})}{match(!\,u : \delta, s, s', \delta', \Theta_{N_S})}$$

$$\frac{match(f_1, f_2, s, \delta, \Theta_{N_S}) \quad match(t_1, t_2, s, \delta, \Theta_{N_S})}{match((f_1, t_1), (f_2, t_2), s, \delta, \Theta_{N_S})}$$

from that of the creating one (with the obvious update for the `self` locality). The type of the new node is obtained from that associated to the locality variable argument of **newloc** and from that of the node where the creating process runs, as specified in the notion of conservative extension. Indeed, the creation of a new node s' modifies the topology of the net, namely the type $\delta_{s''}$ of an old node s'' is "enriched" with the type $s' \mapsto \delta_{s''} \downarrow_p s \mapsto \delta_{s'}$. Thus, any node has a greater type than the one before the inferred reduction. This modification is necessary to allow operations at s'.

Finally, rule (6) deals with process invocation while rule (7) is the standard rule that relates operational semantics and structural congruence.

The pattern matching predicate used in Table 11 is defined in Table 12, and relies on the auxiliary predicate Π_s that we introduce below.

Definition 5.3. Let $\delta^* = \delta[s/u]\{\Theta_{N_S}\}_{s'}$; then predicate $\Pi_s(u, s', \delta, \Theta_{N_S})$ holds if:
— $\delta^* \downarrow_p s \neq \emptyset$ implies $\delta_{s'} \downarrow_p s \sqsubseteq_\Pi \delta^* \downarrow_p s$;
— $s_j \mapsto \pi \mapsto \delta' \in Sub(\delta^*)$ and $\delta' \downarrow_p s \neq \emptyset$ imply $\delta_{s_j} \downarrow_p s \sqsubseteq_\Pi \delta' \downarrow_p s$,
where $Sub(\delta)$ has been introduced in Definition 2.1.

Predicate Π_s is indexed by the site s received in a communication. The parameters of the predicate are the variable that will be bound to s, the site where the operation is invoked, the type of the continuation and the type interpretation function of the net. Predicate $\Pi_s(u, s', \delta, \Theta_{N_S})$ is satisfied when the operations that the continuation process P intends to perform at site s, from any site s'' the process can migrate to, do match the access rights of site s'' over s.

Matching rules ensure that if **read**/**in** looks for sites with type δ and δ'' is the result of interpreting $\delta[s/u]$, then only sites with type δ' such that $\delta'' \preccurlyeq \delta'$ would be accepted; while if **read**/**in** looks for processes with type δ then only processes with type δ' whose interpretation is less than or equal to that of δ would be accepted.

Table 13
An additional type inference rule for Table 7

$\phi \vdash \mathbf{out}(et) : \bot \quad (11)$

6. Types and reductions

In this section, we state and prove a *subject reduction* theorem and a *type safety* theorem. Subject reduction guarantees that well-typedness is an invariant of the operational semantics. Type safety guarantees that well-typed nets are free from run-time errors. Such errors are generated when processes attempt to execute actions that are not permitted by their capabilities. As a result of these two theorems, we have that well-typed nets never encounter run-time errors due to misuse of access rights. In other words, the errors that the KLAIM type system precludes are those due to access right violations.

Before proving these theorems we need to define typing of the auxiliary process $\mathbf{out}(et)$. To this purpose the type inference rule in Table 7 is added to those in Table 13. The rule states that evaluated tuples always have the empty type (since they are represented by passive processes).

To show that well-typedness is an invariant of the operational semantics, we proceed in two steps. First, we prove that well-typedness is preserved under structural congruence; then, we prove that well-typedness is preserved under reduction.

Proposition 6.1. *If N is well-typed and $N \equiv N'$ then N' is well-typed.*

Proof. By easy inspection of the axioms in Table 10. □

Theorem 6.2 (Subject reduction). *If N_S is well-typed and $N_S \rightarrowtail N'_{S'}$ then $N'_{S'}$ is well-typed.*

Proof. By induction on the length of the derivation of $N_S \rightarrowtail N'_{S'}$.

Base step: By case analysis on axioms (1)–(5).

Rule (1). Since no new node is created, we must prove that from $s' = \rho(\ell)$, $et = \mathscr{T}[\![t]\!]_\rho$ and $N_{S'} \| s ::^{\delta}_{\rho} \mathbf{out}(t)@\ell \,.\, P \| s' ::^{\delta'}_{\rho'} P' = N_S$ well-typed it follows that $N_{S'} \| s ::^{\delta}_{\rho} P \| s' ::^{\delta'}_{\rho'} P' \mid \mathbf{out}(et) = N'_S$ is well-typed. The well-typedness hypothesis implies that $\delta_{P'}\{\Theta_{N_S}\}_{s'} \leqslant \delta'$, where $\delta_{P'}$ is a minimal type such that $\phi \vdash P' : \delta_{P'}$. Now, $\mathbf{out}(et)$ has type $\bot$ and by the type inference rule (8) we get $\phi \vdash P' \mid \mathbf{out}(et) : \delta_{P'}$. Since $\Theta_{N_S} = \Theta_{N'_S}$, $\delta_{P'}\{\Theta_{N'_S}\}_{s'} \leqslant \delta'$ directly follows from the well-typedness hypothesis. We are only left to show that if $\phi \vdash P : \delta_P$ with δ_P minimal type then $\delta_P\{\Theta_{N'_S}\}_s \leqslant \delta$. Suppose that $\phi \vdash \mathbf{out}(t)@\ell \,.\, P : \delta_1$. Then, the type inference rule (3) is the last rule used and $\delta_1 \cong \delta_P, \ell \mapsto \{o\} \mapsto \bot$. Since $s' = \rho(\ell)$, the well-typedness hypothesis implies that

$(\delta_P, \ell \mapsto \{o\} \mapsto \bot)\{\Theta_{N_S}\}_s = (\delta_P\{\Theta_{N_S}\}_s, s' \mapsto \{o\} \mapsto \bot) \preccurlyeq \delta$. Then, the thesis follows by the subtyping rule (u/u) and the fact that $\Theta_{N_S} = \Theta_{N'_S}$.

Rule (2). Also in this case no new node is created, then we must prove that from $s' = \rho(\ell)$ and $N_{S'} \parallel s ::^{\delta}_{\rho} \mathbf{eval}(Q)@\ell\,.P \parallel s' ::^{\delta'}_{\rho'} P' = N_S$ well-typed it follows that $N_{S'} \parallel s ::^{\delta}_{\rho} P \parallel s' ::^{\delta'}_{\rho'} P' \mid Q = N'_S$ is well-typed. With respect to the previous case we are left to show that $\delta_Q\{\Theta_{N'_S}\}_{s'} \preccurlyeq \delta'$ where δ_Q is a minimal type such that $\phi \vdash Q : \delta_Q$. Suppose that $\phi \vdash \mathbf{eval}(Q)@\ell\,.P : \delta_1$. Then, the type inference rule (6) is the last rule used and $\delta_1 \cong \delta_P, \ell \mapsto \{e\} \mapsto \delta_Q$, where δ_P is a minimal type such that $\phi \vdash P : \delta_P$. Since $s' = \rho(\ell)$, the well-typedness hypothesis implies that $(\delta_P, \ell \mapsto \{e\} \mapsto \delta_Q)\{\Theta_{N_S}\}_s = (\delta_P\{\Theta_{N_S}\}_s, s' \mapsto \{e\} \mapsto \delta_Q\{\Theta_{N_S}\}_{s'}) \preccurlyeq \delta$. Then, by the subtyping rule (u/u), it follows that $s' \mapsto \{e\} \mapsto \delta_Q\{\Theta_{N_S}\}_{s'} \preccurlyeq \delta$. In particular, this implies that $(s' \mapsto \{e\} \mapsto \delta_Q\{\Theta_{N_S}\}_{s'}) \downarrow_t s' = \delta_Q\{\Theta_{N_S}\}_{s'} \preccurlyeq \delta' = \delta \downarrow_t s'$, where the last equality follows by definition of $\delta \downarrow_t \ell$ and of type of a node. The thesis follows from the fact that $\Theta_{N_S} = \Theta_{N'_S}$.

Rule (3). Again the set of nodes of the net is unchanged and we must prove that from $s' = \rho(\ell)$, $update(\phi, t) \vdash P : \delta_P$ with δ_P minimal type, $match(\mathscr{T}[\![t]\!]_\rho, et, s, \delta_P, \Theta_{N_S})$ and $N_{S'} \parallel s ::^{\delta}_{\rho} \mathbf{in}(t)@\ell\,.P \parallel s' ::^{\delta'}_{\rho} \mathbf{out}(et) = N_S$ well-typed, it follows that $N_{S'} \parallel s ::^{\delta}_{\rho} P[et/\mathscr{T}[\![t]\!]_\rho] \mid s' ::^{\delta'}_{\rho'} \mathbf{nil} = N'_S$ is well-typed. Actually, it suffices to show that from $\delta_1\{\Theta_{N_S}\}_s \preccurlyeq \delta$, where $\phi \vdash \mathbf{in}(t)@\ell\,.P : \delta_1$ with δ_1 minimal type, it follows that $\delta_2\{\Theta_{N'_S}\}_s \preccurlyeq \delta$, where $\phi \vdash P[et/\mathscr{T}[\![t]\!]_\rho] : \delta_2$ with δ_2 minimal type. Since the type inference rules are syntax-driven, rule (5) is the last rule used and $\delta_1 \cong \delta_P, \ell \mapsto \{i\} \mapsto \bot$, where $\delta_P \Downarrow_t u \preccurlyeq \delta_u$ for all $(!\,u : \delta_u) \in \{t\}$. Since $match(\mathscr{T}[\![t]\!]_\rho, et, s, \delta_P, \Theta_{N_S})$ holds, Corollary 4.4 implies $\delta_2 \preccurlyeq \delta_P[\tilde{\ell}/\tilde{\mu}]$, where $[\tilde{\ell}/\tilde{\mu}]$ is the substitution obtained by restricting $[et/\mathscr{T}[\![t]\!]_\rho]$ to locality variables only. Since $\Theta_{N_S} = \Theta_{N'_S}$, we are left to prove that $\delta_P[\tilde{\ell}/\tilde{\mu}]\{\Theta_{N_S}\}_s \preccurlyeq \delta$. The well-typedness hypothesis $\delta_P\{\Theta_{N_S}\}_s \preccurlyeq \delta$ means that the following facts hold:

1. for all $\ell \in S$, $(\delta_P\{\Theta_{N_S}\}_s)\downarrow_p \ell \neq \emptyset$ implies $\delta \downarrow_p \ell \sqsubseteq_\Pi (\delta_P\{\Theta_{N_S}\}_s)\downarrow_p \ell$;
2. for all $\ell' \mapsto \pi'_{\ell'} \mapsto \delta'_{\ell'} \in Sub(\delta_P\{\Theta_{N_S}\}_s)$, for all $\ell \in S$, $\delta'_{\ell'}\downarrow_p \ell \neq \emptyset$ implies $\delta_{\ell'}\downarrow_p \ell \sqsubseteq_\Pi \delta'_{\ell'}\downarrow_p \ell$ and $\delta'_{\ell'}\downarrow_t \ell \preccurlyeq \delta_{\ell'}\downarrow_t \ell$.

Now, it is easy to verify that the following properties hold:

$$(\delta_P[\tilde{\ell}/\tilde{\mu}])\downarrow_p \ell = \begin{cases} \delta_P \downarrow_p \ell & \text{if } \ell \notin \{\tilde{\ell}\}, \\ (\delta_P \downarrow_p \ell) \cup (\delta_P \downarrow_p u) & \text{otherwise;} \end{cases}$$

$$(\delta_P[\tilde{\ell}/\tilde{\mu}])\downarrow_t \ell = \begin{cases} (\delta_P \downarrow_t \ell)[\tilde{\ell}/\tilde{\mu}] & \text{if } \ell \notin \{\tilde{\ell}\}, \\ (\delta_P \downarrow_t \ell)[\tilde{\ell}/\tilde{\mu}], (\delta_P \downarrow_t u)[\tilde{\ell}/\tilde{\mu}] & \text{otherwise;} \end{cases}$$

$$(\delta_P[\tilde{\ell}/\tilde{\mu}])\Downarrow_t \ell = \begin{cases} (\delta_P \Downarrow_t \ell)[\tilde{\ell}/\tilde{\mu}] & \text{if } \ell \notin \{\tilde{\ell}\}, \\ (\delta_P \Downarrow_t \ell)[\tilde{\ell}/\tilde{\mu}], (\delta_P \Downarrow_t u)[\tilde{\ell}/\tilde{\mu}] & \text{otherwise.} \end{cases}$$

By using the above properties, the fact that $\delta_P \Downarrow_t u \preccurlyeq \delta_u$ for all $(!\,u : \delta_u) \in \{t\}$ and the premises of predicate *match* we can conclude that facts 1. and 2. still hold when the type $\delta_P[\tilde{\ell}/\tilde{\mu}]\{\Theta_{N_S}\}_s$ is considered instead of $\delta_P\{\Theta_{N_S}\}_s$. Hence, the thesis follows by the monotonicity of type constructors (Corollary 3.7).

Rule (4). This proof is similar to the one above.

Rule (5). In this case a new node with site s' is created and we must prove that from $s' \notin S'' \cup \{s\}$, $\rho_{s'} = \rho_s[s'/self]$ and $N_{S''} \parallel s ::_{\rho_s}^{\delta_s} newloc(u:\delta).P = N_S$ well-typed with types induced by Λ, it follows that $N_{S''}[\delta^*_{s_i}/\delta_{s_i}]_{s_i \in S} \parallel s ::_{\rho_s}^{\delta^*_s} P[s'/u] \parallel s' ::_{\rho_{s'}}^{\delta^*_{s'}} \mathbf{nil} = N'_{S'}$ is well-typed with types induced by Λ' (where Λ' is the conservative extension of Λ w.r.t. s', δ, u, $\rho_{s'}$ and s, and $\delta^*_{s_i}$ for $s_i \in S'' \cup \{s, s'\}$ are the components of the solution of the system of type equations induced by Λ'). Since the type of any old node is replaced by a greater one, $s' \notin image(\rho_{s_j})$ for any node $s_j ::_{\rho_{s_j}}^{\delta_{s_j}} P_j \in N_S$ and s' has not yet exchanged via communications, we are only left to show that from $\delta_1\{\Theta_{N_S}\}_s \preccurlyeq \delta_s$, where $\phi \vdash \mathbf{newloc}(u:\delta).P : \delta_1$ with δ_1 minimal type, it follows that $\delta_2\{\Theta_{N'_{S'}}\}_s \preccurlyeq \delta^*_s$, where $\phi \vdash P[s'/u] : \delta_2$ with δ_2 minimal type. Since the type inference rules are syntax-driven, rule (7) is the last rule used and $\delta_1 \cong \delta_P, \mathbf{self} \mapsto \{n\} \mapsto \perp$ where $\phi \vdash P : \delta_P$ with δ_P minimal type and $\delta_P \Downarrow_t u = \delta$. By Corollary 4.4, $\delta_2 \preccurlyeq \delta_P[s'/u]$. Now, $\Theta_{N'_{S'}} = \Theta_{N_S}[\rho_{s'}/s']$, hence to prove the thesis it suffices to prove that $\delta_P[s'/u]\{\Theta_{N_S}[\rho_{s'}/s']\}_s \preccurlyeq \delta^*_s$. Moreover, $\delta_P \Downarrow_t u = \delta$ implies $(\delta_P \Downarrow_t u)[s'/u] = \delta[s'/u]$ which, since s' is fresh, implies $\delta_P[s'/u] \Downarrow_t s' = \delta[s'/u]$. From the well-typedness hypothesis, from the definitions of type interpretation and of conservative extension, and by using Corollary 3.7, we have

$$
\begin{aligned}
\delta_P[s'/u]\{\Theta_{N_S}[\rho_{s'}/s']\}_s &= \delta_P[s'/u]\{\delta[s'/u] /\!/ s'\}\{\Theta_{N_S}[\rho_{s'}/s']\}_s \\
&= \delta_P[s'/u]\{\delta[s'/u]\{\Theta_{N_S}[\rho_{s'}/s']\}_{s'} /\!/ s'\}\{\Theta_{N_S}\}_s \\
&\preccurlyeq \delta_P[s'/u]\{\delta^*_{s'} /\!/ s'\}\{\Theta_{N_S}\}_s \\
&\preccurlyeq \delta_P\{\Theta_{N_S}\}_s[s'/s]\{\delta^*_{s'} /\!/ s'\}, \delta_P\{\Theta_{N_S}\}_s\{\delta^*_{s'} /\!/ s'\} \\
&\preccurlyeq \delta_s[s'/s]\{\delta^*_{s'} /\!/ s'\}, \delta_s \\
&\preccurlyeq \delta^*_s.
\end{aligned}
$$

Inductive step: By case analysis on the last operational rule used, namely rule (6) or (7).

Rule (6). By induction since, by Corollary 4.4, $\phi \vdash A\langle \tilde{P}, \tilde{\ell}, \tilde{e}\rangle : \delta[\tilde{\ell}/\tilde{\mu}]$, where $\phi \vdash A : \delta$ and $A(\tilde{X} : \tilde{\delta}, \tilde{\mu} : \tilde{\delta}, \tilde{x}) \stackrel{\text{def}}{=} P$, and $\phi \vdash P[\tilde{P}/\tilde{X}, \tilde{\ell}/\tilde{\mu}, \tilde{e}/\tilde{x}] : \delta'$ with δ' minimal type imply $\delta' \preccurlyeq \delta[\tilde{\ell}/\tilde{\mu}]$.

Rule (7). By standard inductive arguments and by Proposition 6.1. □

We now introduce the notion of run-time error and prove type safety. Run-time errors are defined in Table 14 in terms of the predicate $N \xrightarrow{s} error$ that holds true when within the net N a process P at site s attempts to perform an action that is not allowed by its capabilities. Let us recall that the type of a site codifies the policy of access rights and that $\delta \downarrow_p s'$ denotes the access rights of s over s', for s and s' sites and δ type of s.

We rely on the following notation. Given an action a ($\mathbf{in}(t)@\ell_1$, $\mathbf{out}(t)@\ell_2, \ldots$), $cap(a)$ denotes the corresponding capability $(i, o, \ldots)$ and $loc(a)$ denotes the locality name in a ($\ell_1, \ell_2, \ldots$). Thus, $loc(\mathbf{out}(t)@\ell) = \ell$ and similarly for the other actions apart

Table 14
Run-time error

$$(act)\quad \frac{\delta \downarrow_p \rho(loc(a)) \not\sqsubseteq_\Pi \{cap(a)\}}{s ::^\delta_\rho a.P \xrightarrow{s} error}$$

$$(ide)\quad \frac{s ::^\delta_\rho P[\tilde{P}/\tilde{X}, \tilde{\ell}/\tilde{u}, \tilde{e}/\tilde{x}] \xrightarrow{s} error}{s ::^\delta_\rho A\langle \tilde{P}, \tilde{\ell}, \tilde{e}\rangle \xrightarrow{s} error} \qquad A(\tilde{X} : \tilde{\delta}, \tilde{u} : \tilde{\delta}, \tilde{x}) \stackrel{\text{def}}{=} P$$

$$(par)\quad \frac{N \xrightarrow{s} error}{N \parallel N' \xrightarrow{s} error}$$

$$(str)\quad \frac{N \equiv N' \qquad N' \xrightarrow{s} error}{N \xrightarrow{s} error}$$

for **newloc** for which we let $loc(\mathbf{newloc}(u:\delta)) = \mathbf{self}$. We write $\not\sqsubseteq_\Pi$, $\delta \not\leqslant \delta'$ and $N \stackrel{s}{\not\to} error$ for the negation of $\sqsubseteq_\Pi$, $\delta \leqslant \delta'$ and $N \xrightarrow{s} error$, respectively.

Theorem 6.3 (Type safety). *If N is well-typed then, for every site s, $N \stackrel{s}{\not\to} error$.*

Proof. By induction on length of the derivation of $N \xrightarrow{s} error$, we prove that $N \xrightarrow{s} error$ implies that N is not well-typed.

The base step is when rule (*act*) of Table 14 is the only rule used. In this case we have that $N = s ::^\delta_\rho a.P$ and $s ::^\delta_\rho a.P \xrightarrow{s} error$ because $\delta \downarrow_p \rho(loc(a)) \not\sqsubseteq_\Pi \{cap(a)\}$. The thesis obviously holds when $a.P$ is not typable in the type context ϕ. Hence, assume that a type δ' exists such that $\phi \vdash a.P : \delta'$ (without loss of generality, let δ' be a minimal type for $a.P$). We are left to show that $\delta'\{\Theta_{N_s}\}_s \not\leqslant \delta$. By the type inference rules (3)–(7) in Table 7 and by rule ($\mapsto/\mapsto$) of subtyping, we have that $loc(a) \mapsto \{cap(a)\} \mapsto \bot \leqslant \delta'$. Hence, by Definition 4.9, we have that $(loc(a) \mapsto \{cap(a)\} \mapsto \bot)\{\rho\} \leqslant \delta'\{\Theta_{N_s}\}_s$. The fact that $\delta \downarrow_p \rho(loc(a)) \not\sqsubseteq_\Pi \{cap(a)\}$ implies that $\rho(loc(a)) \mapsto \{cap(a)\} \mapsto \bot = (loc(a) \mapsto \{cap(a)\} \mapsto \bot)\{\rho\} \not\leqslant \delta$. The thesis now follows because otherwise we would have $\delta'\{\Theta_{N_s}\}_s \leqslant \delta$ and, by transitivity, $(loc(a) \mapsto \{cap(a)\} \mapsto \bot)\{\rho\} \leqslant \delta$.

We now consider the inductive step. We proceed by case analysis on the last rule applied in the derivation of $N \xrightarrow{s} error$.

(*ide*) Assume that $A(\tilde{X} : \tilde{\delta}, \tilde{u} : \tilde{\delta}, \tilde{x}) \stackrel{\text{def}}{=} P$; we have $s ::^\delta_\rho A\langle \tilde{P}, \tilde{\ell}, \tilde{e}\rangle \xrightarrow{s} error$ because $s ::^\delta_\rho P[\tilde{P}/\tilde{X}, \tilde{\ell}/\tilde{u}, \tilde{e}/\tilde{x}] \xrightarrow{s} error$. Again, the thesis obviously holds when $A\langle \tilde{P}, \tilde{\ell}, \tilde{e}\rangle$ is not typable in the type context ϕ. Hence, assume that there exists a type δ_1 such that $\phi \vdash A\langle \tilde{P}, \tilde{\ell}, \tilde{e}\rangle : \delta_1$. We show that $\delta_1\{\Theta_{N_s}\}_s \not\leqslant \delta$. By the type inference rules (9) and (10), $\phi \vdash A\langle \tilde{P}, \tilde{\ell}, \tilde{e}\rangle : \delta_1$ implies that there exists a type δ_2 such that $\phi[\widetilde{\delta_X}/\tilde{X}][\delta_2/A] \vdash P : \delta_2, \delta_1 = \delta_2[\tilde{\ell}/\tilde{u}]$ and, for each $P_i \in \{\tilde{P}\}$, $\phi \vdash P_i : \delta_i$ and $\delta_i \leqslant \delta_{X_i}$. Hence, by Corollary 4.4, we can derive that there exists a type δ_3 such that $\phi[\delta_2/A] \vdash P[\tilde{P}/\tilde{X}, \tilde{\ell}/\tilde{u}, \tilde{e}/\tilde{x}] : \delta_3$ and $\delta_3 \leqslant \delta_2[\tilde{\ell}/\tilde{u}]$. By induction, we may assume that $s ::^\delta_\rho P[\tilde{P}/\tilde{X}, \tilde{\ell}/\tilde{u}, \tilde{e}/\tilde{x}] \xrightarrow{s} error$ implies that $s ::^\delta_\rho P[\tilde{P}/\tilde{X}, \tilde{\ell}/\tilde{u}, \tilde{e}/\tilde{x}]$ is not

well-typed, i.e. that $\delta_3\{\Theta_{N_S}\}_s \not\preccurlyeq \delta$. This together with the fact that $\delta_3 \preccurlyeq \delta_2[\tilde{\ell}/\tilde{u}] = \delta_1$ and with Definition 4.9 imply the thesis.

(*par*) Notice that if N' is not well-typed then $N \| N'$ is not well-typed. Then, it suffices to apply the inductive hypothesis.

(*str*) Apply the inductive hypothesis and Proposition 6.1. □

Since well-typedness is preserved along sequences of reduction steps, we can conclude that well-typed nets never generate run-time errors during their evolution.

Corollary 6.4. *Let $\rightarrowtail^*$ be the reflexive and transitive closure of $\rightarrowtail$. If N is well-typed and $N \rightarrowtail^* N'$ then there is no site s such that $N' \xrightarrow{s}$ error.*

Proof. The proof proceeds by induction on the length of $N \rightarrowtail^* N'$. The base step is Theorem 6.3, the inductive step follows from Theorems 6.2 and 6.3. □

Remark 6.5. A choice we had to face when designing the type system was that of fixing the type to be associated to newly created sites. Our choice was to allow programmers to impose type constraints on the newly created site by explicitly restricting the type of the node where the creating process is running.

Here we briefly discuss the impact of a different choice that does not allow programmers to impose type restrictions when creating new sites.

The type inference rule for **newloc**(u) becomes

$$\frac{\Gamma \vdash P : \delta}{\Gamma \vdash \mathbf{newloc}(u).P : (\delta, \mathbf{self} \mapsto \{n\} \mapsto \bot)} \qquad (7')$$

and no check is performed on the kind of the operations that P intends to perform at u; the rule only extends the type of P at **self** with n. The notion of extension needs now to be modified to allow dynamically created nodes to inherit the access rights of the creating one. An extension Λ' of a Λ now only depends on $s' \notin S$ and $s \in S$, and the last item of Definition 5.1 becomes

- for all $s_1 \in S$, $\Lambda'(s_1)(s') = \Lambda(s_1)(s)$, $\Lambda'(s')(s_1) = \Lambda(s)(s_1)$ and $\Lambda'(s')(s') = \Lambda(s)(s)$.

Moreover, any extension is conservative. The operational rule for **newloc** does not formally change, but the new definition of extension leads to greater types of the dynamically created nodes.

7. Programming examples and paradigms

In this section we provide a complete programming example in KLAIM. Then, we illustrate some basic programming paradigms for mobile code applications.

7.1. Distributed information retrieval

Distributed information retrieval applications gather information from a set of sources distributed over a network; the nodes to be visited may be determined either statically or dynamically. This kind of application is a paradigmatic example for mobile computation. Here, we present a KLAIM solution of a distributed information retrieval problem and discuss some of the access control issues there involved.

Let us assume that *strings*, denoted by sequences of characters between quotation marks, are basic values. Consider a user process that needs information about a data of which he only has a key represented, say, by "*item*", to be used for searching in a database distributed over the network.

In our solution, the distributed database is modelled by located tuple spaces and it is assumed that each local database (a tuple space) reachable from ℓ_{item} (the starting point of the search which is known by the user process) contains either a tuple of the form ("*item*", v) with the required information v, or a tuple of the form ("*item*", s_{next}), with the site of the next node to search.

The *user process UP* asks for the execution at ℓ_{item} of the mobile *gatherer* agent, G, which travels across the nodes looking for tuples containing information associated to "*item*". This agent takes as parameters the key "*item*" and a freshly created locality u_r that represents the private address where *UP* would like to receive the result of the search.

The user process *UP* can be programmed in KLAIM as follows:

$$UP = \mathbf{newloc}(u_r : \bot).\mathbf{eval}(G\langle \text{"}item\text{"}, u_r\rangle)@\ell_{item}.\mathbf{in}(!\,x)@u_r.P$$

The agent G is programmed as follows:

$$\begin{aligned} G(x, u : \bot) = \ & \mathbf{read}(x, !\,u' : \delta')@self.\mathbf{eval}(G\langle x, u\rangle)@u'.\mathbf{nil} \\ & \mid \mathbf{read}(x, !\,y)@self.\mathbf{out}(y)@u.\mathbf{nil} \end{aligned}$$

The site dynamically created by *UP* is accessible only via u_r; hence, if it is assumed that P never communicates the value of u_r, then only processes *UP* and G can access this site. To guarantee that for continuing the search the gatherer agent receives only sites with appropriate rights, it suffices to choose δ' such that $\delta_G[u_r/u] \preccurlyeq \delta'$.

If process P has type δ_P, then the inferred types of *UP* and G are

$$\begin{aligned} \delta_{UP} &= \mathbf{self} \mapsto \{n\} \mapsto \bot, u_r \mapsto \{i\} \mapsto \bot, \ell_{item} \mapsto \{e\} \mapsto \delta_G[u_r/u], \delta_P \\ \delta_G &= \mu v.(\mathbf{self} \mapsto \{r\} \mapsto \bot, u \mapsto \{o\} \mapsto \bot, u' \mapsto \{e\} \mapsto v). \end{aligned}$$

Let us now see how types can be used to enforce access control policies by considering a net with three sites s, s_1 and s_2, with types δ_s, δ_{s_1} and δ_{s_2}. Assume that s_1 contains the tuple ("*item*", s_2), s_2 contains the tuple ("*item*", v), and the user process *UP* is at site s and $\rho_s(\ell_{item}) = s_1$. Then, the relevant part of the net is

$$s ::^{\delta_s}_{\rho_s} UP \parallel s_1 ::^{\delta_{s_1}}_{\rho_{s_1}} \mathbf{out}(\text{"}item\text{"}, s_2) \parallel s_2 ::^{\delta_{s_2}}_{\rho_{s_2}} \mathbf{out}(\text{"}item\text{"}, v).$$

Checking whether the net is well-typed requires determining the interpretation of δ_{UP} with respect to the allocation environments and comparing the resulting type with δ_s. The interpretation of δ_{UP} is

$$s \mapsto \{i,n\} \mapsto \bot, s_1 \mapsto \{e\} \mapsto (s_1 \mapsto \{r\} \mapsto \bot, s \mapsto \{o\} \mapsto \bot), [\delta_P]$$

where $[\delta_P]$ denotes the interpretation of δ_P. For guaranteeing well-typedness, we should make sure that

$$(s \mapsto \{i,n\} \mapsto \bot, s_1 \mapsto \{e\} \mapsto (s_1 \mapsto \{r\} \mapsto \bot, s \mapsto \{o\} \mapsto \bot), [\delta_P]) \preccurlyeq \delta_s$$

which depends on $(s_1 \mapsto \{r\} \mapsto \bot, s \mapsto \{o\} \mapsto \bot) \preccurlyeq \delta_{s_1}$ and $[\delta_P] \preccurlyeq \delta_s$.

The type δ' of the variable u' used by the agent G when reading continuation sites is used statically (by the type inference system) to check whether $\delta_G[u_r/u] \preccurlyeq \delta'$, and dynamically (by the matching predicate) to select the appropriate tuple. For instance, the gatherer agent G can travel from s_1 to s_2 only if the tuple ("*item*",s_2) can be selected; this amounts to requiring that the interpretation of $\delta'[s_2/u']$ is less than or equal to δ_{s_2}.

7.2. Code mobility

Mobile code applications are applications running over a network whose distinctive feature is the exploitation of forms of "mobility". Below we first show how to model code-on-demand, then describe how one can use it together with (non-obvious) type structures to implement "safe" remote evaluation.

Code-on-demand: A component of an application running over a network on a given node, can dynamically download some code from a remote node and link it to perform a given task.

To download code that respects certain type constraints δ and is stored in a tuple at a remote tuple space l, action $\mathbf{read}(!X:\delta)@l.X$ can be used. The downloaded code is checked for access violations at run time by the pattern-matching mechanism. The typing rules for **read** and **in** are designed in such a way that any downloaded code whose type is a subtype of δ will not violate type correctness.

Remote evaluation: Any component of a networking application can invoke services from other components by transmitting both the data needed to perform the service and the code that describes how to perform the service.

To transmit both code P and data v at the locality l of the server, action $\mathbf{out}(\mathbf{in}(!y)@l.A\langle y\rangle, v)@l$, where $A(x) \stackrel{\text{def}}{=} P$, can be used. Here, we assume that the server adopts the following (code-on-demand) protocol

$$\mathbf{in}(!X:\delta, !x)@self.\mathbf{out}(x)@self.X.$$

To prevent "damages" from P, the server may mount and execute code P only if $[\delta_P]$, the type of the code P when interpreted at the server's site, is such that $[\delta_P] \preccurlyeq [\delta]$, where $[\delta]$ is the interpretation of δ at the server's site.

Type δ may give only minimal access permissions on the server's site, for instance, only the capability of reading some resources and giving back the results of the execution. In this case, $[\delta]$ is of the form

$$s \mapsto \{r\} \mapsto \bot, s_0 \mapsto \{o\} \mapsto \bot, s_1 \mapsto \{o\} \mapsto \bot, \ldots s_n \mapsto \{o\} \mapsto \bot,$$

where s_i, $i = 0, \ldots, n$, are the sites with the rights of invoking server's facilities, and s is the server's site.

Notice, however, that this does not prevent P from visiting other sites. In particular, agent P may be programmed in such a way that after having performed the required elaboration, it transmits code Q at the locality l_i (the logical name of site s_i):

$$P \stackrel{\text{def}}{=} \mathbf{read}(!x)@l.\mathbf{out}(op(x,y))@\mathbf{self}.\mathbf{out}(Q)@l_i.\mathbf{nil}.$$

It is immediate to see that, if we assume that the client is allocated at site s_0 and that P has type $\delta_P = l \mapsto \{r\} \mapsto \bot, self \mapsto \{o\} \mapsto \bot, l_i \mapsto \{o\} \mapsto \bot$, then $[\delta_P] = s \mapsto \{r\} \mapsto \bot$, $s_0 \mapsto \{o\} \mapsto \bot, s_i \mapsto \{o\} \mapsto \bot$. Hence, $[\delta_P] \preccurlyeq [\delta]$. However, code Q is only stored in the tuple space at s_i: no new thread of execution is activated at site s_i. Before being executed code Q must be read and verified (dynamic type checking). Therefore, process P cannot activate a *Trojan horse* at the remote site s_i.

8. Detailed proofs

In this section we present the proofs of some syntactical properties of the KLAIM type system, that have been skipped or just sketched in the earlier sections.

Proposition 8.1 (Proposition 3.3). *For any type δ there is a canonical type δ' such that $\delta \cong \delta'$.*

Proof. A simple (innermost) procedure to reduce δ to δ' can be defined as follows. First, by (8), erase all μ-binders that bind no variables and, by (9), reduce sequences of μ-binders, that is rewrite $\mu v_1 \ldots \mu v_n.\delta$ as $\mu v.\delta[v/v_1, \ldots, v/v_n]$ and erase any $\delta_i = v$ in $\mu v.\delta_1, \ldots, \delta_n$. Then, reduce union types via a two-step procedure. By (1)–(5), erase components that are $\bot$ or repetition (possibly use (7)), and absorb in $\top$ other components. By (7) and, possibly, (8) and (9) fold all μ operators as much as possible, so that no components of the union type is a μ-type and sequences of μ-types are reduced to only one μ-type. Finally, once a type of the form $\mu v.\phi_1, \ldots, \phi_n$ or $\phi_1, \ldots, \phi_n$ is obtained, such that all ϕ_i are canonical, then use (6) to rewrite $\phi_1, \ldots, \phi_n$ so that $L(\phi_i) \cap L(\phi_j) = \emptyset$. □

Proposition 8.2 (Proposition 3.6). *Let δ', δ'' and δ''' be canonical forms. If $\delta' \preccurlyeq \delta''$ and $\delta'' \preccurlyeq \delta'''$ are provable, then $\delta' \preccurlyeq \delta'''$ is provable.*

Proof. By induction on the derivation of $\delta' \preccurlyeq \delta''$. The base step is trivial since, whenever $\delta' \preccurlyeq \delta''$ is an axiom, then either $\delta' = \bot$, or $\delta'' = \top$ (then $\delta''' = \top$), or $\delta' \cong \delta''$

and canonical forms are considered as equivalent up to folding/unfolding. The inductive step is proved by case analysis on the last rule used for deriving $\delta' \leqslant \delta''$. For each case, consider all possible subcases in relation to the last rule used for deriving $\delta'' \leqslant \delta'''$. The proof follows from the inductive hypothesis by relying on the following two properties:

(i) if $\mathscr{D}$ is a derivation of $\ell \mapsto \pi \mapsto \delta \leqslant \delta^*$ (for some δ^* different from $\top$ and from $\ell \mapsto \pi \mapsto \delta$ up to $\cong$), then $\delta^* = \mu v.\delta_1, \ldots, \delta_n$ (or $\delta^* = \delta_1, \ldots, \delta_n$), $\exists i : 1 \leqslant i \leqslant n$ such that $\delta_i = \ell \mapsto \pi_i \mapsto \delta_i'$, and $\pi_i \sqsubseteq_{\Pi} \pi$ and $\delta \leqslant \delta_i'[\delta^*/v]$ (or $\delta \leqslant \delta_i'$) are already proved in $\mathscr{D}$;

(ii) if $\mathscr{D}$ is a derivation of $\delta^* \leqslant \ell \mapsto \pi \mapsto \delta$ (for some δ^* different from $\bot$ and from $\ell \mapsto \pi \mapsto \delta$ up to $\cong$), then $\delta^* = \mu v.\ell \mapsto \pi_1 \mapsto \delta_1$ (or $\delta^* = \ell \mapsto \pi_1 \mapsto \delta_1$), and $\pi \sqsubseteq_{\Pi} \pi_1$ and $\delta_1 \leqslant \delta[\delta^*/v]$ (or $\delta_1 \leqslant \delta$) are already proved in $\mathscr{D}$. □

Theorem 8.3 (Theorem 3.8). *The subtyping relation* $\leqslant$ *is decidable.*

Proof. Consider the following algorithm for proving $\delta_1 \leqslant \delta_2$.

1. If δ_1 (δ_2) is $\bot$ or $\top$, then apply axioms (ax$\bot$) or (ax$\top$) and stop.
2. If $\delta_1 \cong \delta_2$ then stop (by axiom (eq)).
3. If δ_1 or δ_2 is a union type, then apply (u/u) until the remaining subgoals do not involve union types.
4. If both δ_1 and δ_2 are arrow types, apply ($\mapsto/\mapsto$).
5. If one of δ_1 and δ_2 is an arrow type and the other is a μ-type, then proceed in the following way. Without loss of generality, we may assume that δ_1 is $\ell \mapsto \pi \mapsto \delta'$ and δ_2 is $\mu v.\phi_1, \ldots, \phi_n$ (the converse is similar). Let us consider the unfolded form of δ_2. If $\ell \notin L(\phi_i[\delta_2/v])$ for all i ($1 \leqslant i \leqslant n$) then fail. Otherwise, by canonicity, there is a single ϕ_i such that $\phi_i[\delta_2/v]$ is of the form $\ell \mapsto \pi_i \mapsto \delta_i$; then check $\pi_i \sqsubseteq_{\Pi} \pi$ and $\delta' \leqslant \delta_i^*$, where δ' is already in canonical form and δ_i^* is a canonical form of δ_i.
6. If both δ_1 and δ_2 are μ-types, apply (μ/μ) and compare their bodies.

Soundness of the decision algorithm follows from the fact that steps 3, 4 and 6 have a direct correspondent in the subtyping rules, while step 5 corresponds to applying either ($\mapsto/v$) or ($v/\mapsto$), plus ($\mapsto/\mapsto$). We are left to prove that the algorithm terminates. This proof proceeds by induction on the cardinality of $Sub(\delta_1) \cup Sub(\delta_2)$, the set of subterms of the types δ_1 and δ_2 (by definition, this cardinality cannot become smaller than 2). Since $\sqsubseteq_{\Pi}$ on polarities and $\cong$ on types are decidable, termination of the decision algorithm follows from the fact that steps 1 and 2 do not generate subgoals, while steps 3–6 generate new subgoals with a smaller set of subterms. □

For proving existence of the minimal type, we need some technical preliminaries.

Definition 8.4. A *type scheme* σ is a pair $\langle \mu v.\delta, \{v_1 = \delta_1, \ldots, v_n = \delta_n\} \rangle$ where $n \geqslant 1$ and $v, v_1, \ldots, v_n$ are all the free type variables that can occur in δ and in each type δ_i. The *solution* of σ, $\mu(\sigma)$ is the type $\mu v.\delta[\delta_1^*/v_1, \ldots, \delta_n^*/v_n]$, where $\delta_1^*, \ldots, \delta_n^*$ is the solution of the system of type equations in the type scheme.

Proposition 8.5. *Given* $\sigma = \langle \mu v.\delta, \{v_1 = \delta[\tilde{\ell_1}/\tilde{u_1}], \ldots, v_n = \delta[\tilde{\ell_n}/\tilde{u_n}]\}\rangle$ *and* $\sigma' = \langle \mu v.\delta', \{v_1 = \delta'[\tilde{\ell_1}/\tilde{u_1}], \ldots, v_n = \delta'[\tilde{\ell_n}/\tilde{u_n}]\}\rangle$, *if* $\delta \leqslant \delta'$ *then* $\mu(\sigma) \leqslant \mu(\sigma')$.

Proof. By definition, $\delta \leqslant \delta'$ implies $\delta[\tilde{\ell_i}/\tilde{u_i}] \leqslant \delta'[\tilde{\ell_i}/\tilde{u_i}]$ for all $1 \leqslant i \leqslant n$. This, by Corollary 3.7, implies $\mu(\sigma) \leqslant \mu v.\delta[\delta_1^*/v_1, \ldots, \delta_n^*/v_n]$ where $\delta_1^*, \ldots, \delta_n^*$ is the solutions of the set of equations $\{v_1 = \delta'[\tilde{\ell_1}/\tilde{u_1}], \ldots, v_n = \delta'[\tilde{\ell_n}/\tilde{u_n}]\}$. Now, $\delta \leqslant \delta'$ implies $\mu v.\delta[\delta_1^*/v_1, \ldots, \delta_n^*/v_n] \leqslant \mu v.\delta'[\delta_1^*/v_1, \ldots, \delta_n^*/v_n] = \mu(\sigma')$. Then, $\mu(\sigma) \leqslant \mu(\sigma')$ follows by Proposition 3.6. □

Lemma 8.6. *Let* $\mathscr{D}$ *be a derivation of* $\Gamma, \tilde{X} : \tilde{\delta}, A : \delta' \vdash P : \delta'$ *in the proof system, where* $A(\tilde{X} : \tilde{\delta}, \tilde{u} : \tilde{\delta}, \tilde{x}) \stackrel{\text{def}}{=} P$ *and no process identifier other than* A *occurs in* P. *Then, there exists a minimal type* δ *such that*

(i) $\Gamma, \tilde{X} : \tilde{\delta}, A : \delta \vdash P : \delta$ *is derivable,*

(ii) $\delta \leqslant \delta'$.

Proof. The assumption that no process identifier other than A occurs in P implies that rule (9) is not used in $\mathscr{D}$ but in the last step. Then the minimal type δ can be constructed in the following way. Assume that the process identifier A has type v, where v is a fresh type variable. For any occurrence of $A\langle \tilde{P}, \tilde{\ell}, \tilde{e}\rangle$ in P assume $A\langle \tilde{P}, \tilde{\ell}, \tilde{e}\rangle : v[\tilde{\ell}/\tilde{u}]$ without applying the substitution. As a consequence, the type derivation for P relies on a set of hypotheses (i.e. type constraints involving v) which must be checked later.

By considering the rules of Table 7 except for (9), it is easy to verify that our type derivations have the form

$$\Gamma, \tilde{X} : \tilde{\delta}, A : v \vdash P : \delta''$$

where v or $v[\tilde{\ell}/\tilde{u}]$ can occur in δ''. Moreover, rule (9) requires $v = \delta''$ and then the type derived for P (the body of A) is the solution $\mu(\sigma)$ of the type scheme $\sigma = \langle \mu v.\delta_1, \{v_1 = \delta_1[\tilde{\ell_1}/\tilde{u_1}], \ldots, v_n = \delta_1[\tilde{\ell_n}/\tilde{u_n}]\}\rangle$ where δ_1 is obtained by replacing in δ'' any occurrence of $v[\tilde{\ell_i}/\tilde{u_i}]$ with v_i (v_i fresh). This gives a derivation of the minimal type $\delta = \mu(\sigma)$ for P, which is sound only if subtyping constraints involving v, not yet evaluated, are satisfied. These constraints can now be checked since types, not type schemata, are dealt with. The checking will be successful by construction if all the intentions of P on $\tilde{u}$ (registered in the minimal type) agree with the type declared for the parameters; otherwise P is untypable and the algorithm fails.

For point (ii), because of the deterministic construction of the type for P except for the case $A(\tilde{X} : \tilde{\delta}, \tilde{u} : \tilde{\delta}, \tilde{x}) \stackrel{\text{def}}{=} P$, we have that δ' must codify at least all the intentions of P which are in δ_1 minimal type). Hence, δ' is such that $\delta' = \mu(\sigma')$ where $\sigma' = \langle \mu v.\delta_2, \{v_1 = \delta_2[\tilde{\ell_1}/\tilde{u_1}], \ldots, v_n = \delta_2[\tilde{\ell_n}/\tilde{u_n}]\}\rangle$, and $\delta_1 \leqslant \delta_2$. Therefore, by Proposition 8.5, $\delta = \mu(\sigma) \leqslant \mu(\sigma') = \delta'$. □

Since our types are monotonic with respect to type substitution, the previous lemma can be immediately generalized to the case where process identifiers other than A occur in the body P of the definition of A.

Corollary 8.7. *Let* $\Gamma, \tilde{X} : \tilde{\delta}, A : \delta', A_1 : \delta_1, \ldots, A_n : \delta_n \vdash P : \delta'$ *where* $A(\tilde{X} : \tilde{\delta}, \tilde{u} : \tilde{\delta}, \tilde{x}) \stackrel{\text{def}}{=} P$ *and all process identifiers occurring in* P *are in* $\{A, A_1, \ldots, A_n\}$. *Let* $\delta_1, \ldots, \delta_n$ *such that, for any* $A_i(\tilde{X}_i : \tilde{\delta}, \tilde{u}_i : \tilde{\delta}, \tilde{x}_i) \stackrel{\text{def}}{=} P_i$ $(1 \leqslant i \leqslant n)$,

$$\Gamma, \tilde{X}_i : \tilde{\delta}, A_1 : \delta_1, \ldots, A_i : \delta_i' \ldots, A_n : \delta_n \vdash P_i : \delta_i'$$

implies $\delta_i \leqslant \delta_i'$. *Then, there exists a minimal type* δ *for* P *such that* $\Gamma, \tilde{X} : \tilde{\delta}, A : \delta, A_1 : \delta_1, \ldots, A_n : \delta_n \vdash P : \delta$ *and* $\delta \leqslant \delta'$.

We can now prove existence of the minimal type for a given P typable in Γ.

Theorem 8.8 (Theorem 4.5). *If* $\Gamma \vdash P : \delta'$ *for some* δ', *then there exists a minimal type* δ *such that* $\Gamma \vdash P : \delta$ *and* $\delta \leqslant \delta''$ *for all* δ'' *such that* $\Gamma \vdash P : \delta''$.

Proof. The minimal type of P can be defined, in a trivial way, by structural induction on P, from the minimal types of its subterms. The only interesting case concerns occurrences of process definitions in P. For any process definition, the algorithm defined in the proof of Lemma 8.6 is used to obtain the minimal type. The statement can now be proven by induction on the length of the derivation of $\Gamma \vdash P : \delta''$.

Base step: All cases are trivial but rule (9), for which use Lemma 8.6.

Inductive step: Apply directly the inductive hypothesis. It suffices to notice that in each inference rule the type of the term in the consequent is greater than the type of the subterms in the antecedent and that type constructors are monotonic. In the case of rule (9), the proof follows from Corollary 8.7. □

9. Concluding remarks

We have developed a type system which formalizes access control restrictions of programs written in KLAIM. Type information is used to specify access rights and execution privileges, and to detect violations of these policies. The implementation of the type inference system for X-KLAIM (the prototype implementation of KLAIM) is in progress; it will help us also to assess our design choices.

Recently, distributed variants of the π-calculus have been introduced (e.g. Ambient Calculus [13] and Dπ [25]) as foundational calculi for network programming. Syntactically, these calculi are very different from KLAIM, however simple variations of KLAIM types can be applied to them to specify and enforce access control policies.

We plan to extend the type system by introducing types for tuples (record types), notions of multi-level security (by structuring localities into levels of security) and public or shared keys to model dynamic transmission of access rights. Ideas could also be borrowed from the spi-calculus [2], a concurrent calculus obtained by adding public-key encryption primitives to the π-calculus [29], and from the SLam calculus [24], another calculus where information about direct/indirect producers and consumers are associated to data.

Another direction for future research is considering "open" systems. In fact, our type system can safely deal with new processes landing on existing nodes, but it does not consider partially specified nets. An enrichment of types is then needed to specify the permissions granted to "unspecified" sites. Then, the choice has to be faced whether this enrichment should be specified at the level of single nodes or at the level of nets. Interfacing nets would naturally fit with extensions of our framework to hierarchical nets that would be beneficial also for more structured access controls. An alternative approach to deal with open systems could also be that of relaxing the static type checking phase by not requiring well-typedness of the whole net (this corresponds to the fact that only the typed sites can be trusted) while increasing the run-time type checking phase, e.g. agents migrating from untyped (i.e. untrusted) sites must be dynamically typechecked. This is the approach followed in [34].

Type systems have been used also for other calculi of mobile processes. Among those reminiscent of ours, although not addressing security issues, we mention the work of Pierce and Sangiorgi [32]. They develop a type system for the π-calculus using channels types to specify whether channels are used to *read* or to *write*. This type system has been extended in [26] by associating *multiplicities* to types for stating the number of times each channel can be used. The type system of [32] has been also generalized by Sewell [35] to capture locality of channel names and by Boreale and Sangiorgi [10] to trimmer bisimulation proofs.

Only recently attempts have been made to characterize security properties in terms of formal type systems. A type system for the spi-calculus has been developed by Abadi [1] to guarantee secrecy of cryptographic protocols. Abadi and Stata [3] have used type rules to specify and verify correctness of Java Bytecode Verifier. Hennessy and Riely [25] have introduced a type system for the language Dπ. This work is similar to ours (types are abstraction of process behaviours and access rights violations are type errors), but the technical developments are quite different; resources are channels and types describe permissions to use channels. Moreover, access rights are fixed irrespectively of the localities where processes themselves are executed. In [34], the type system of [25] has been improved for considering nets where sites can set up malicious agents that do not respect the rules on the use of resources. Cardelli and Gordon [14] have introduced a type system for mobile ambients [13] that controls the type of the values exchanged among administrative domains (ambients) so that the communication of values cannot cause run-time faults. Volpano and Smith have developed type systems to ensure secure information flow (noninterference) for both a sequential procedural language [38] and for a multithreaded imperative language [39]. Boudol [11] has used types to abstract from terms the possible sequences of interactions and the resources used by processes. Necula [30] has introduced an approach to ensure correctness of mobile code with respect to a fixed safety policy, where code producers provide the code with a proof of correctness that code consumers check before allowing the code to execute. Vitek and Castagna [37] have proposed a language-based approach, relying on powerful mobility and protection primitives, rather than on type systems, for secure Internet programming. Bodei et al. [9] have proposed an alternative

approach for the analysis of security and of information flow, that relies on static analysis techniques.

Acknowledgements

We would like to thank Luca Cardelli, Mario Coppo and Mariangiola Dezani–Ciancaglini for useful discussions and comments, and Marco Di Costanzo for signaling an imprecision in a previous version of the paper. We are grateful to the anonymous referees for several suggestions that greatly helped us in revising the paper and to Daniel Le Metayer for the way he handled our submission.

References

[1] M. Abadi, Secrecy by typing in cryptographic protocols, in: M. Abadi, M. Ito (Eds.), Theoretical Aspects of Computer Software (TACS'97), Proc., Lecture Notes in Computer Science, vol. 1281, Springer, Berlin, 1997, pp. 611–638.

[2] M. Abadi, A.D. Gordon, A calculus for cryptographic protocols: the spi calculus, Inform. and Comput. 148(1) (1999) 1–70.

[3] M. Abadi, R. Stata, A type system for Java Bytecode subroutines, Proc. ACM Symp. on Principles of Programming Languages, ACM Press, New York, 1998, pp. 149–160. ACM Trans. Programm. Languages Systems, to appear.

[4] R. Amadio, An asynchronous model of locality, failure and process mobility, in: D. Garlan, D. Le Metayer (Eds.), COORDINATION'97, Proc., Lecture Notes in Computer Science, vol. 1282, Springer, Berlin, 1997, pp. 374–391.

[5] R. Amadio, L. Cardelli, Subtyping recursive types, ACM Trans. Programm. Languages Systems 15(4) (1993) 575–631.

[6] R. Amadio, S. Prasad, Localities and failures, in: P.S. Thiagarajan (Ed.), Foundations of Software Technology and Theoretical Computer Science (FSTTCS'94), Proc., Lecture Notes in Computer Science, vol. 880, Springer, Berlin, 1994, pp. 205–216.

[7] A. Arnold, J. Gosling, The Java Programming Language, Addison-Wesley, Reading, MA, 1996.

[8] L. Bettini, R. De Nicola, G. Ferrari, R. Pugliese, Interactive mobile agents in X-KLAIM, Proc., IEEE 7th Internat. Workshop on Enabling Technologies: Infrastructure for Collaborative Enterprises, IEEE Computer Society Press, Silverspring, MD, 1998.

[9] C. Bodei, P. Degano, F. Nielson, H.R. Nielson, Control flow analysis for the π-calculus, in: D. Sangiorgi, R. de Simone (Eds.), Concurrency Theory (CONCUR'98), Proc., Lecture Notes in Computer Science, vol. 1466, Springer, Berlin, 1998, pp. 611–638.

[10] M. Boreale, D. Sangiorgi, Bisimulation in naming-passing calculi without matching, Proc. 13th IEEE Symp. on Logic in Computer Science (LICS '98), IEEE Computer Society Press, Silverspring, MD, 1998, pp. 165–175.

[11] G. Boudol, Typing the use of resources in a concurrent calculus, in: R.K. Shyamasundar, K. Ueda (Eds.), Advances in Computing Science (ASIAN'97), Proc., Lecture Notes in Computer Science, vol. 1345, Springer, Berlin, 1997, pp. 239–253.

[12] M. Brandt, F. Henglein, Coinductive axiomatization of recursive type equality and subtyping, in: P. de Groote, J.R. Hindley (Eds.), Third Internat. Conf. on Typed Lambda Calculi and Applications, Lecture Notes in Computer Science, vol. 1210, Springer, Berlin, 1997, pp. 63–81.

[13] L. Cardelli, A. Gordon, Mobile ambients, in: M. Nivat (Ed.), Foundations of Software Science and Computation Structures (FoSSaCS'98), Proc., Lecture Notes in Computer Science, vol. 1378, Springer, Berlin, 1998, pp. 140–155.

[14] L. Cardelli, A. Gordon, Types for mobile ambients, Proc. ACM Symp. on Principles of Programming Languages, ACM Press, New York, 1999, pp. 79–92.

[15] N. Carriero, D. Gelernter, Linda in context, Comm. ACM 32(4) (1989) 444–458.
[16] N. Carriero, D. Gelernter, J. Leichter, Distributed data structures in Linda, Proc. ACM Symp. on Principles of Programming Languages, ACM Press, New York, 1986, pp. 236–242.
[17] R. De Nicola, G. Ferrari, R. Pugliese, Coordinating mobile agents via blackboards and access rights, in: D. Garlan, D. Le Metayer (Eds.), Coordination Languages and Models (COORDINATION'97), Proc., Lecture Notes in Computer Science, vol. 1282, Springer, Berlin, 1997, pp. 220–237.
[18] R. De Nicola, G. Ferrari, R. Pugliese, KLAIM: a Kernel language for Agents interaction and mobility, IEEE Trans. Software Eng. 24(5) (1998) 315–330.
[19] R. De Nicola, R. Pugliese, A process algebra based on Linda, in: P. Ciancarini, C. Hankin (Eds.), COORDINATION'96, Proc. Lecture Notes in Computer Science, vol. 1061, Springer, Berlin, 1996, pp. 160–178. (Theoret. Comput. Sci., to appear.)
[20] M. Dezani-Ciancaglini, U. de Liguoro, A. Piperno, A filter model for concurrent λ-calculi, SIAM J. Comput. 27(5) (1998) 1376–1419.
[21] C. Fournet, G. Gonthier, J.-L. Lévy, L. Maranget, D. Rémy, A calculus of mobile agents, in: U. Montanari, V. Sassone (Eds.), CONCUR'96, Proc., Lecture Notes in Computer Science, vol. 1119, Springer, Berlin, 1996, pp. 406–421.
[22] D. Gelernter, Generative communication in Linda, ACM Trans. Programm. Languages Systems 7(1) (1985) 80–112.
[23] D. Gelernter, N. Carriero, S. Chandran et al., Parallel programming in Linda, Proc. IEEE Internat. Conf. on Parallel Programming, IEEE Computer Society Press, Silverspring, MD, 1985, pp. 255–263.
[24] N. Heintz, J.G. Riecke, The SLam calculus: programming with secrecy and integrity, Proc. ACM Symp. on Principles of Programming Languages, ACM Press, New York, 1998, pp. 365–377.
[25] M. Hennessy, J. Riely, Resource access control in systems of mobile agents, Proc. Internat. Workshop on High-Level Concurrent Languages, Electronic Notes in Theoretical Computer Science, vol. 16, Elsevier, Amsterdam, 1998.
[26] N. Kobayashi, B. Pierce, D. Turner, Linearity and the π-calculus, Proc. ACM Symp. on Principles of Programming Languages, ACM Press, New York, 1996.
[27] D. Kozen, J. Palsberg, M. Schwartzbach, Efficient recursive subtyping, Mathematical Structures in Computer Science 5 (1995) 113–125.
[28] R. Milner, Communication and Concurrency, Prentice-Hall Int., Englewood Cliffs, NJ, 1989.
[29] R. Milner, J. Parrow, D. Walker, A calculus of mobile processes (Parts I and II), Inform. and Comput. 100 (1992) 1–77.
[30] G. Necula, Proof-carrying code, Proc. ACM Symp. on Principles of Programming Languages, ACM Press, New York, 1997, pp. 106–119.
[31] U. Nestmann, B.C. Pierce, Decoding choice encodings, in: U. Montanari, V. Sassone (Eds.), CONCUR'96, Proc., Lecture Notes in Computer Science, vol. 1119, Springer, Berlin, 1996, pp. 179–194.
[32] B. Pierce, D. Sangiorgi, Typing and subtyping for mobile processes, Math. Struct. Comput. Sci. 6(5) (1996) 409–454.
[33] R. Pugliese, Semantic theories for asynchronous languages, Ph.D. Thesis VIII-96-6, Univ. di Roma "La Sapienza", Dip. Scienze dell'Informazione, 1996.
[34] J. Riely, M. Hennessy, Trust and partial typing in open systems of mobile agents, Proc. ACM Symp. on Principles of Programming Languages, ACM Press, New York, 1999.
[35] P. Sewell, Global/local subtyping and capability inference for a distributed π-calculus, in: K.G. Larsen, S. Skyum, G. Winskel (Eds.), Internat. Colloquium on Automata, Languages and Programming (ICALP'98), Proc., Lecture Notes in Computer Science, vol. 1443, Springer, Berlin, 1998, pp. 695–706.
[36] R. Statman, Recursive types and the subject reduction theorem, Technical Report 94-164, Carnegie Mellon University, 1994.
[37] J. Vitek, G. Castagna, Towards a calculus of secure mobile computations, Proc. Workshop on Internet Programming Languages, Chicago, 1998.
[38] D. Volpano, G. Smith, A typed-based approach to program security, in: M. Bidoit, M. Dauchet (Eds.), Theory and Practice of Software Development (TAPSOFT'97), Proc., Lecture Notes in Computer Science, vol. 1214, Springer, Berlin, 1997, pp. 607–621.
[39] D. Volpano, G. Smith, Secure information flow in a multi-threaded imperative language, Proc. ACM Symp. on Principles of Programming Languages, ACM Press, New York, 1998, pp. 355–364.